THE PROGRAMMER'S
Essential OS/2
HANDBOOK

REGULAR
$24.95
DISCOUNT
$12.48

DAVID E. CORTESI

Save yourself the time and trouble of manual file entry. . .

Order *The Programmer's Essential OS/2 Handbook* Appendix Disk.

All the programs in this book are available on disk (MS-DOS format), with full source code. You'll find: files for both **C and Pascal** that declare OS/2 system calls and data structures; exploratory programs that allow the inquisitive programmer to probe the system and find out what it really does in any circumstance; device monitor programs that demonstrate multithread monitoring of the keyboard, the printer, and the mouse; an example dynamic link library that shows how to add a new feature to the system; and much more.

To order, return this postage-paid self-mailer with your payment of $25, plus sales tax if you are a California resident, to: M&T Books, 501 Galveston Drive, Redwood City, CA 94063. Or, call toll-free 800-533-4372 (in CA 800-356-2002) between 8 a.m. and 5 p.m., Monday through Friday, Pacific Standard Time. Ask for Item # 000-1. Please specify 5-1/4" or 3-1/2" OS/2 format.

YES! Please send me the OS/2 Appendix Disk for $25 _____

California residents add applicable sales tax _____% _____

TOTAL _____

Format: _____5-1/4" _____3-1/2"

_____ Check enclosed. Make payable to M&T Books.

Charge my ____ VISA ____ MasterCard ____ American Express

Card # _____ Exp. date _____

Name _____

Address _____

City _____ State _____ Zip _____

40001

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT 871 REDWOOD CITY, CA

POSTAGE WILL BE PAID BY ADDRESSEE

M&T BOOKS
501 Galveston Drive
Redwood City, CA 94063

PLEASE FOLD ALONG LINE AND STAPLE OR TAPE CLOSED

The Programmer's Essential OS/2 Handbook

The Programmer's **Essential OS/2** Handbook

David E. Cortesi

M&T Publishing, Inc.
Redwood City, California

M&T Books
A Division of M&T Publishing, Inc.
501 Galveston Drive
Redwood City, CA 94063

M&T Books
General Manager, Ellen Ablow
Project Manager, Michelle Hudun
Editor, Sally Brenton
Cover Designer, Frank Pollifrone
Illustrator, Lynn Sanford

© 1988 M&T Publishing, Inc.

Printed in the United States of America
First Edition published 1988

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without prior written permission from the Publisher. Contact the Publisher for information on foreign rights.

Library of Congress Cataloging-in-Publication Data

Cortesi, David E.
The programmer's essential OS/2 handbook.

Includes indexes.
1. MS OS/2 (Computer operating systems) I. Title.
QA76.76.063C68 1988 005.4'465 88-3559
ISBN 0-934375-82-8 (book)
ISBN 0-934375-89-5 (book/disk)
ISBN 0-55851-000-1 (disk)

92 91 90 89 88 5 4 3 2 1

The following terms are trademarks of the International Business Machines Corporation:

- IBM PC
- OS/2
- PC/AT
- PS/2
- PC/DOS

These terms are trademarks of the Microsoft Corporation:

- CodeView
- MASM
- MS DOS and MS-DOS
- MS OS/2
- Microsoft

Limits of Liability and Disclaimer of Warranty

The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

How to Order the Appendix Disk

All the programs in this book are available on disk (MS-DOS format) with full source code. The disk price is $25.00. California residents must add the appropriate sales tax.

Order by sending a check made payable to **M&T Books**, or your credit card number and expiration date, to:

OS/2 Appendix Disk
M&T Books
501 Galveston Drive
Redwood City, CA 94063

Or, you may order by calling our toll-free number between 8 A.M. and 5:00 P.M. Pacific Standard Time: 800/533-4372 (800/356-2002 in California).

Contents

Preface 13
What's Essential? 13
What's Here 14
What's Left Out 14
How Errors will be Handled 15
Acknowledgments 16

1: System Overview 17
1.1 The System Hardware 17
1.2 Fundamental Concepts 18
1.3 CPU Modes 21
1.4 Storage Management 23
1.5 Disk Storage 23
1.6 Static and Dynamic Linking 24
1.7 The Execution Environment 26
1.8 Exeception Handling 27
1.9 Asynchronous Processing 28
1.10 Video Display Control 31
1.11 Keyboard Input 32
1.12 Mouse Input 32
1.13 Printer Output 33
1.14 Device I/O and Device Monitors 33
1.15 System Extensions 34

2: The Hardware Environment 37
2.1 Fundamental Ideas 37

2.2 The Machine Registers....40
2.3 The "Memory Model"....50
2.4 The Address Space....54
2.5 Privilege Levels....66

3: Linking....71
3.1 Linking and Loading....71
3.2 An Overview of Linking....72
3.3 Controlling the Linker....95
3.4 Summary....117

4: Loading....119
4.1 The Executable File....119
4.2 The Loading Process....125
4.3 The Use of Loaded Modules....126
4.4 Initializing Dynamic Link Packages....131
4.5 Run-Time Dynamic Linking....132

5: The Execution Environment....139
5.1 The State of the System....139
5.2 National Language Support....158
5.3 Unplanned Events....162
5.4 The Message Services....174

6: Managing Subordinate Programs....181
6.1 Subordinate Processes....182
6.2 Subordinate Sessions....190
6.3 Restricted System Calls....195
6.4 Data Into Code....196

7: Storage Allocation....197
7.1 Moving Segments....197
7.2 Swapping and Discarding Segments....198
7.3 Storage Allocation....200
7.4 Sharing Allocated Segments....205

8: Files 209
8.1 Volumes, Directories and Paths 209
8.2 Date Words and Time Words 212
8.3 The Attributes of a File 213
8.4 File Handles 214
8.5 File System Operations 215
8.6 Directory Operations 219
8.7 The *DosOpen* Call 228
8.8 Closing a File 235
8.9 Changing a File's Size 235
8.10 Operations on Handles 236
8.11 Input and Output 238

9: Asynchronous Processing 249
9.1 Threads 249
9.2 Coordinating Asynchronous Threads 258
9.3 Critical Sections 260
9.4 Semaphores 261
9.5 Timer Services 268
9.6 A Design Study 269
9.7 Queues 280

10: Video Output 285
10.1 Display Fundamentals 285
10.2 Screen Groups 287
10.3 Stream Output to the Display 289
10.4 Direct Display Output 294
10.5 Graphics Mode Operations 313

11: Keyboard Input 325
11.1 Is a Keyboard Available? 325
11.2 Keyboard Decoding 326
11.3 Keyboard Sharing 328
11.4 The Keyboard Status and Modes 331
11.5 Reading a Sequence of Keystrokes 333
11.6 Direct Keyboard Input 333
11.7 Keystroke Translation and Code Pages 337
11.8 Program-Driven Translation 338

11.9 Replacing Keyboard Functions 341

12: Mouse Input 351
12.1 How the Mouse Works 351
12.2 Opening the Mouse 354
12.3 Preparing the Mouse 356
12.4 Using the Mouse 363
12.5 Closing the Mouse 367
12.6 Replacing Mouse Functions 367

13: Device Drivers 371
13.1 Device Drivers 371
13.2 Device Driver Operations 373
13.3 Fake Device Drivers 374
13.4 The IOCTL Mechanism 374
13.5 Generic I/O Control 375

14: Serial Operations 379
14.1 Use of the Serial Ports 379
14.2 The RS-232 Interface 380
14.3 The Serial Port Device Driver 391

15: Whole-Disk Operations 419
15.1 Opening a Disk as a File 420
15.2 Locking a Disk 420
15.3 Querying Disk Information 421
15.4 Handling Removable Volumes 424

16: Device Monitors 425
16.1 What a Monitor Is 425
16.2 What a Monitor Does 426
16.3 Structure of a Monitor 426
16.4 Keystroke Monitors 429
16.5 Mouse Monitors 431
16.6 Print Monitors 432

Collected Error Codes 435

Index to System Calls 449

General Index 453

System Calls 467

Appendix A
Collected Data Structures 697

Preface

This book is for people who write programs for OS/2. It aims to give you the OS/2 technical info that will let you write efficient, reliable applications in C, Pascal, or assembler, organized in a way that lets you find what you need quickly.

Some technical books are put together at third hand from written documents and interviews with developers, but not this one. The only interview I conducted was a six-month-long dialogue with OS/2 itself, during which I poked into every corner of the system using Microsoft's C and Pascal compilers and the excellent CodeView debugger as probes. The draft manuals provided by Microsoft were my guides on a long expedition into the heart of a rich, exciting new system, and this is a report to my fellow programmers on what I discovered.

What's Essential?

My plan was to make an OS/2 reference so complete and so handy that you'd find it essential to your work. That's the first of three meanings for the word *Essential* in the title.

Its second meaning is that, since there wasn't a prayer of covering OS/2 technology completely, the book contains only the things I judged essential to writing good applications. It omits enough material for another volume as fat as this one; I hope it didn't omit something that you deem essential.

OS/2 is so rich that even its essence is massive. From the start it was clear that my only hope of producing a book you could lift, at a price you would pay, in a time the market would allow, lay in boiling the text itself down to the essence. There was no room for tutorial, digression, or helpful repetition. This is a terse, dense book; some people may find it forbidding. I hope its meaty content will compensate for its demanding narrative style.

What's Here

The first part of the book is organized as an outline. The table of contents shows its structure down to the third level of headings. If you are new to OS/2, start with the overview in the first topic; it defines terms essential to all the other topics.

The topics that follow cover the main features of OS/2, explaining its facilities, putting them in context, and showing how they can be used together. The nature of OS/2's architecture is such that it's hard to discuss any feature without referring to every other feature. These topics reflect that; they stand alone, but they are linked by a web of cross-references. It almost doesn't matter where you start, because if you follow out all the references you will visit every other topic anyway.

At the back of the book there's a reference to the 200-odd system calls, one or two pages each, ordered alphabetically. The interface to each function is shown in a graphic form that can be translated to any programming language, and also in C and Pascal declarations. The action of each function is stated in language as concise and unambiguous as I could make it.

These reference pages are not mere compilations from the system documentation! I've personally exercised these system calls; that's how I was able to list what even the official reference books leave out: the error code numbers that the system returns for common mistakes.

In the middle, between the two main parts of the book, you'll find separate indexes to discussion topics, to system calls, and to error codes.

Finally, there's the Disk Appendix: a diskette of software I wrote while exploring OS/2. There are the C header files and Pascal Include files to declare the system functions, a selection of the more interesting test programs so you can explore the system calls the way I did, and a few interesting utility programs. The Disk Appendix is available separately from the publisher in two formats, a 1.2MB, five-inch floppy and a 740KB, 3.5-inch diskette.

What's Left Out

The Presentation Manager will be an important part of OS/2. Programs that interact with their user will benefit from using it, and interactive programs that don't use it will look old-fashioned. But the Presentation Manager couldn't be included in this book. There wasn't room for it, for one thing. For another, as

we go to press the Presentation Manager exists only as a three-volume specification, so I couldn't write it up from experience.

The topic of writing a device driver had to be omitted. This is a serious omission, and I hope to remedy it in a future edition. Not many people write device drivers but I would like to see the knowledge of how to do it spread beyond a small coterie of wizards.

The DosPTrace function has been omitted, so if you plan to write a debugger you'll need another book in addition to this one. Also omitted were a family of system functions that can't be used by any program except the Session Manager (also known as the Program Selector), so this book won't help you write a new Session Manager.

The only programming utility discussed is LINK. As I write, it is not clear how programs like LIB, MAKE, MKMSGF, or BIND will be priced and distributed. LINK is the only tool that's common to all languages, and it has suffered all its life from weak documentation, so I covered it thoroughly, but I left the rest of your tools to be documented by their vendors.

How Errors will be Handled

Errors are inevitable (though not forgivable) in a work of this scope. They will be fixed in later printings as they are found, but that won't help the brave purchaser of this early printing. If before November 30th, 1988 you will send a self-addressed envelope with postage for two ounces to the address below, I will send you information on errors so you can correct your book.

I would be delighted to hear from you. I enjoy hearing about your experiences, opinions, and suggestions; and though I enjoy them less I value your reports of errors and oversights even more. If you are the first to report a substantive error, point out an important omission, or make a useful suggestion that is later adopted, you will receive credit in a later edition. Furthermore, M&T Publishing will present a free copy of that later edition to the people credited in it.

It's great to be contacted by readers, but I'd really rather they didn't do it by telephone. Correspondence by mail is not only cheaper, it's more likely to get a considered response. To be eligible for the offers above, please write to David E. Cortesi, M&T Books, 501 Galveston Drive, Redwood City, CA 94063.

Acknowledgments

This book owes its existence in part to the generosity and dedication of:

Erv Wittman, who gave me moral support and material help when I needed it, as well as the opportunity for my first look at OS/2.

Microsoft Corporation, which contributed an OS/2 Software Developer's Kit and a beta-test copy of Pascal 4.0.

The *Microsoft personnel* who bravely responded to the 95 or so TARs I submitted on DIAL, some of which were less temperate in tone and less accurate in content than I might wish in hindsight.

Club American Technologies, Inc., of Fremont, California, for the use of an AT-compatible and technical help in making it run OS/2.

1

System Overview

OS/2 is an *operating system*, that is, a program that controls and manages the hardware of a computer. The main job of an operating system is to allocate the use of the hardware among many *application programs*, which use the hardware to do productive work for a human being. It must share out the hardware in such a way that the applications don't interfere with one another or overwrite one another's data.

If it works well, the operating system will do all this without harming the efficiency of the application programs, and without getting in the way of the *operator*, the human being who supervises the computer and decides what work it will be applied to.

OS/2 succeeds quite well at these things. It loads multiple independent applications and shares out the system hardware among them so that they appear to run concurrently. OS/2 also succeeds at the even more difficult job of running an MS-DOS application concurrently with other programs.

1.1 The System Hardware

OS/2 requires an Intel 80286 or 80386 CPU and at least two megabytes (abbreviated MB) of RAM storage in order to run. These silicon chips are usually found in an IBM Personal System/2 (model 50 and up), or in an IBM PC/AT, or in a computer that is compatible with one of these.

The system almost always contains a fixed disk of at least 20MB capacity, and almost always contains at least one diskette drive. The system always contains a display screen and a keyboard, and often has a mouse.

These are the hardware resources that OS/2 manages and shares out among concurrent applications. The method of sharing is different for each kind of resource. The CPU is time-shared among loaded programs. Storage is divided into segments and allocated using the addressing hardware. The disks are managed as an MS-DOS system with network-style file sharing. The most elaborate mechanisms are involved in sharing the use of the keyboard, mouse and display screen among all the programs.

1.2 Fundamental Concepts

Three fundamental concepts govern the allocation of resources: the *screen group*, the *process*, and the *thread*.

1.2.1 The Screen Group

A *screen group* represents the hardware used by the operator: the display screen, the keyboard, and the mouse. OS/2 pretends that there are sixteen sets of this hardware: sixteen groups of screen, keyboard and mouse. In reality it supports only one set of hardware, but it arranges things so that there appear to be sixteen sets—sixteen *virtual* displays each with a virtual keyboard and a virtual mouse.

A program is usually associated with a particular screen group and uses that screen group's virtual hardware to interact with the operator. At any time just one of the screen groups is attached to the real hardware. That screen group is said to be *in the foreground*. A program associated with that group uses the real hardware when it reads the keyboard, gets a mouse event, or writes to the screen.

The remaining screen groups are disconnected from the hardware, but they still seem functional to the programs associated with them. These screen groups (and by extension, the associated programs) are said to be *in the background*. Nobody ever presses a key on a keyboard that's in the background; a background mouse never generates an event; output to a background display goes into a storage buffer and will only become visible when the screen group comes to the foreground.

References: display output ch. 10; keyboard input ch. 11; mouse input ch. 12

1.2.2 The Session and Session Manager

A screen group with its associated program (or programs; there may be more than one) is called a *session*. Sessions are managed by a program called the Session Manager. It is also called the Program Selector, from the title at the head of the menu it displays.

The Session Manager creates and destroys screen groups, and associates programs to them. The operator chooses which screen group will be in the foreground using *Session Manager hot keys*. The Session Manager gets special services from the keyboard device driver that tell it when either the Control-Escape or Alt-Escape key combination is pressed. On the first, it redisplays the Program Selector menu so that the operator may choose an existing screen group or start a new one. On Alt-Escape, the Session Manager switches the hardware to the next screen group in rotation.

Reference: creating sessions 6.2

1.2.3 The Process

A *process* is a loaded program, together with all the resources it owns. A process is a static object whose primary function is to own resources: such as segments of storage, open files, and other things that a program can request from OS/2.

The principal thing a process owns is an *address space*, which is the set of storage segments that it can address. But a process also owns a table of open files; this table determines the meaning of the *file handles* used within that process. Other things that are unique to each process include:

- A write-verify switch setting
- A current drive letter and current directory path
- A segment of "environment" strings

One of the important features of OS/2 is that one process may create another, and the subordinate or child process inherits some of its parent's resources.

References: starting a process 4.3; creating child processes 6.1

1.2.4 The Thread

The *thread* is the executable unit, the entity that is allowed to share the use of the CPU. A thread occupies and animates a process. It's important to understand the process as a static thing; as a collection of storage segments and invisible system control blocks. A process is to the threads that animate it as a playing field is to the teams that play on it.

What a thread really consists of is a set of registers. That's all, just a set of machine registers. Of course, the registers include the Local Descriptor Table Register that selects the Local Descriptor Table of the process that the thread occupies; that in turn gives it access to a address space. And the registers include the Code Segment and Instruction Pointer that define for the thread its "next instruction" address. And they include the Stack Segment and Stack Pointer, so the thread has a "top of stack;" the stack defines a history of procedure calls and their local variables and return addresses.

The storage for saving a thread's registers is furnished by OS/2 apart from the address space of the process.

References: creating and managing threads 9.1; first thread in a process 4.3.1

1.2.5 Screen Group, Process, Thread

These three concepts are separate, and you should have no preconceptions about how they may be combined. Ordinarily a screen group will be used by just one process, but a process can create other processes and these descendants share the parent's screen group. There's nothing to stop a process from sharing its screen group with dozens of children. Ordinarily a process will be animated by only one thread, but one thread can create another, and there's nothing to stop a program from having dozens of concurrent threads in its process. (There's a system limit on threads, set at configuration time, but it's quite generous.)

One process can create others, and by the inheritance rules it can determine what its children will see for file handles and environment variables. This opens endless pathways to complex, multi-process applications. Furthermore, a process can create new sessions, that is, start its child processes in independent screen groups. Further still, a process can switch its child screen groups in and out, which opens the way for a program to set up as a subordinate version of the Session Manager.

1.3 CPU Modes

The 80286 CPU has two radically different modes of operation. The predecessor to OS/2, Microsoft's MS-DOS and IBM's PC-DOS, kept the CPU in *Real Mode*. OS/2 runs it as much as possible in *Protect Mode*. The character of Protect Mode has a lot to do with the character of OS/2 itself.

1.3.1 Real Mode

The *address space* of a program is the set of all storage locations that the program can access in any way, whether as code or data. In Real Mode, all programs share a common, one-megabyte address space. Parts of this space are dedicated by hardware to display buffers and read-only program text (the "ROM BIOS" chips). MS-DOS occupied part of the remaining space, and the MS-DOS application program occupied the remainder.

OS/2 usually gives over the first 640 kilobytes (KB) of storage to MS-DOS programs running in a simulated MS-DOS environment. Most MS-DOS applications can run in this mode. As far as one of these programs can tell, it is running under MS-DOS with network support installed. In fact the DOS kernel in low storage is an OS/2 simulation of the MS-DOS system kernel. Many system functions that MS-DOS would have handled are in fact passed to the real OS/2 kernel for execution. The network file-sharing support is necessary because all files are potentially shared, not with a network server, but with other applications that are invisible to the MS-DOS program.

The simulated MS-DOS system is called the Compatibility Process, or colloquially the *DOS Box*. It is an optional feature and can be disabled by a configuration statement. When it is present, OS/2 treats it as a single process running a single thread under a single screen group. The Compatibility screen group can be switched to the foreground or background like any screen group. When it is in the foreground, OS/2 time-shares the CPU between the single Compatibility thread and all other threads. When it is in the background, it is frozen and does not run. Threads in other screen groups continue to share the CPU when they are in the background. The Compatibility thread is frozen because OS/2 cannot prevent a Real-mode program from modifying the real display buffers, keyboard, or mouse out of turn.

Implementing the DOS Box is one of the cleverest technical feats OS/2 performs. For example, the 80286 CPU has no designed method of switching back to Real Mode once it enters Protect Mode. OS/2 accomplishes it by exploiting a glitch,

the Triple Fault. For another example, the addressing hardware of an IBM PC/AT has to operate differently in Real Mode and the difference is controlled by a switch built into the keyboard controller—which means that the switch back to Protect Mode at the end of a time slice may be delayed by incoming keystrokes.

Despite its technical cleverness, and despite its importance as a bridge from MS-DOS to OS/2, the Compatibility process is not really significant to a software designer. Any clean MS-DOS application will run in it, but the interesting applications are the ones that exploit the full power of OS/2. They are the focus of this book.

References: real-mode address space 2.4.1; timesharing and dispatching 9.1.6

1.3.2 Protect Mode

The rules of addressing undergo a drastic change when the CPU switches into protect Mode. Now all references to storage are indirect, via the entries in two *descriptor tables* prepared by the operating system. The *Global Descriptor Table* (GDT) is used by all processes, while each process has a unique *Local Descriptor Table* (LDT). Each descriptor in a table describes a segment of storage: its starting address, its length, and what use may be made of it—whether or not it may be executed as code, used as data, or modified. Furthermore, each segment has an associated *privilege level* that determines which programs may reference it at all.

In protect Mode a program's address space contains exactly and only what is described by the GDT and the program's own LDT, so the operating system has total control over what the program may touch, modify, or execute.

OS/2 exploits the Protect Mode hardware to load many different programs into storage simultaneously, and to arrange that none can harm or snoop into another's affairs or affect the operating system. It also uses the hardware to allow programs to share segments of code (it tries never to load more than one segment of code no matter how many programs use it) and to share segments of data in controlled, voluntary ways. Thus, while there can be only one application in the emulated MS-DOS area, there may be many Protect Mode applications in the system concurrently, and OS/2 can distribute the hardware fairly among them all.

References: protect-mode address space 2.4.2

1.4 Storage Management

Because of the Protect Mode hardware, OS/2 can treat storage as a general pool from which it can allocate segments to different programs. The first 640 kilobytes (KB) of storage are usually given over to the Compatibility Process. The display buffer and ROM segments that are dedicated by hardware are set aside under privileged descriptors that only OS/2 can refer to. Compatibility programs can refer to these areas since they fall into the address space of a Real Mode program, but to Protect Mode programs they simply don't exist. (A program may gain access to the physical display buffer if necessary for graphics output.)

OS/2 allocates blocks of storage to programs upon request. It carves a block out of the common pool and inserts a descriptor for it in the requestor's LDT. Storage can be had in blocks of any size, including so called "huge" segments larger than 64KB. Blocks of 64KB or less can be suballocated in the manner of a C storage arena using system services.

The Protect Mode descriptor for each segment contains a present bit. If the bit is false, a hardware trap takes place when the segment is used. OS/2 uses this to implement *segment swapping*, which is a method of overbooking storage. When storage space is required but not available, it chooses the least recently used segments, ensures that they have been copied to disk, and then discards them to make room. When (and if) the program that owns a discarded segment tries to use it next, a hardware trap occurs. OS/2 makes room (possibly discarding other segments in order to do so) and reloads the swapped segments from disk. Then it restarts the interrupted program, which is never aware of the delay.

> **References:** access to physical display buffer 10.5.2; storage allocation 7.3; segment swapping 7.2; suballocation 7.3.5

1.5 Disk Storage

The management of disk space is easy to deal with: OS/2 stores data on disk identically to MS-DOS 3.3. The directory structures, rules for naming files, and means of partitioning a fixed disk are all identical. In its first release, the MS-DOS restriction to 32 megabytes per disk partition also applies; this is supposed to change for both MS-DOS and OS/2 in future releases.

Applications normally access the disk by opening named files. The OS/2 services for file access are more extensive, more flexible, more convenient and faster than those of MS-DOS. Support for locating files using environment variables like PATH is integrated into the file support. Any application may schedule

asynchronous file I/O, so that it can overlap processing with input or output. Files can be shared among concurrent processes, and any span of data from a single byte up to the whole file can be locked against concurrent updates.

It is also possible for a program to open an entire disk volume, so that the entire disk surface, directory structure and all, can be treated as a file. This powerful feature must obviously be used with care, but it permits the creation of disk patch and unerase utilities.

> **References:** file access ch. 8; search of path list 8.6.5.1; file locking 8.11.8; whole-disk I/O ch. 15

1.6 Static and Dynamic Linking

OS/2 stores executable programs in files of type .EXE, just like MS-DOS, but the file format is much more complicated. The file format accommodates three objectives: compatibility with MS-DOS, support for Protect mode, and dynamic linking.

1.6.1 The OS/2 Executable File

The OS/2 executable file contains the MS-DOS executable file as a subset, so that a single file can contain an MS-DOS program and an OS/2 program. Each system loads the program that it wants, and neither system wastes time reading the other system's program data.

1.6.1.1 Family Applications

This clever arrangement opens the way to *family* programs. A family program is one written for OS/2, but which restricts itself to the subset of OS/2 function that can be simulated under MS-DOS. The program's executable file contains an MS-DOS program called the stub loader, as well as the OS/2 executable program.

When a family program is run under OS/2, only the OS/2 executable is loaded. When it runs under MS-DOS, the MS-DOS system loads only the stub loader and starts it. That program knows of the existence of the OS/2 program further on in the .EXE file. It loads the family program, resolves its OS/2 dynamic links, and emulates OS/2 functions for it.

1.6.1.2 The Protect-Mode Program

For MS-DOS a linked program was a monolithic binary image; that is, every part of the program runs together into a sequence of bytes. For Protect mode, the boundaries between individual segments must be preserved along with their types, so that a Local Descriptor Table can be built to describe them as code or data. The OS/2 executable file consists of a header followed by a series of segment images, each in a subfile with its own relocation information.

OS/2 supports *dynamic linking*, in which some of a program's references to external procedures may be left unresolved until the program is loaded (or even later). To support this the executable file allows for a table of character-string names of external procedures. The relocation information following a segment's image can refer to this table, and the system will bind the reference to the named procedure as the program is loaded.

1.6.1.3 Dynamic Link Packages

The object of a dynamic link—the entry point that resolves a program's external reference—is an *exported* name in a *dynamic link library*. A dynamic link library is an executable file, prepared exactly like a program by compiling and linking. However, a dynamic link library cannot execute as an independent program; it consists of a package of subroutines that are exported to the *client* programs that refer to them.

OS/2 loads dynamic link libraries as required, when it loads a client program that needs one. Only one copy of a dynamic link library is loaded, no matter how many clients may be using it concurrently.

Since a dynamic link package is linked and loaded just like a program, it is free to refer to other dynamic link packages, and they to others, and so forth. The OS/2 kernel itself is presented as if it were a dynamic link package named DOSCALLS; and major subsystems (keyboard, video, mouse) are in fact dynamic link libraries.

> **References:** executable file format 4.1; linking dynamic link package 3.3.4.1, 3.3.4.7.2; loading and initializing dynamic link packages 4.4

1.6.2 The OS/2 Linker

The new executable file requires a new linker to build it. There are many more options for the formation of programs, so while the Linker is compatible with the Link command of MS-DOS, it supports many more options. It takes a *definition file* of commands in which the programmer may describe the program down to the level of individual segments.

The Linker is central to all programming efforts on OS/2, and the longest single topic in this book is devoted to a cursory survey of its features.

> **Reference:** link usage ch. 3

1.7 The Execution Environment

When a program is loaded, it is converted into a process with an address space that includes its loaded segments plus an environment segment. The process is animated with a single thread that begins execution at the defined program entry point.

The process can gain access to a wealth of information about the system and its configuration using a family of system calls that either return information or give access to read-only segments of system data. The environment segment contains two blocks of strings: one block that defines variables in the conventional *name=value* form, and the other contains the program's executable filename and its command-line parameters.

The process receives three open file handles, for standard input, standard output, and error reports (the standard printer file handle of MS-DOS is not supported by OS/2). These handles will have been prepared by the process's parent. They will usually be opened to the screen group's keyboard and screen—but the parent process may set them up to any source or sink of data that it pleases. If the parent is the command shell, the standard handles may have been opened to disk files or to pipes, which are buffered communication paths shared with other processes.

The process can create its own children, and control the meaning of their handles and the contents of the environment segment they inherit.

> **References:** program entry 4.3; environment segment 5.1.5; file handles 8.4; preparing handles for a child process 6.1.4; creating pipes 6.1.5.

1.8 Exception Handling

A process can set up its own handlers for three kinds of exceptional events.

1.8.1 Software Signals

The first class of events are software signals generated by other processes. Three of these signals are benign, and are used for communication between processes. However, the Control-C and Control-Break keyboard signals are also converted into software signals. The default handlers for these signals convert them into a third signal, the termination signal. Other processes may also issue the termination signal—the software version of pronouncing a curse! The default handler for the termination signal cancels the process.

A process may set up individual handler procedures for any of these six signals. The handlers for the communication signals may take any desired action. The handlers for the Control-C, Control-Break, and termination signals may ignore them or may take an orderly, planned shutdown.

Handler policies may be nested, so that the code of a subsystem can set up its own temporary handler and then revert to its caller's handler when it exits.

1.8.2 Hardware Traps

A second class of events are certain hardware traps. A process can set up its own handler for such events as dividing by zero, executing an invalid operation code, or the detection of an error by the 80287 Numeric Coprocessor. As with software signals, handler policies can be nested so that subsystems can install their own and then revert to their callers' policy when finished.

1.8.3 Exit Lists

For every process the system maintains an Exit list, a list of procedures that are to be called when the process terminates regardless of the cause of termination. There are system calls to add a routine to the exit list and to remove it, so that a linked subsystem could install an exit handler while it is working for a process and remove it when the process is done with its services.

References: software signals 5.3.1; hardware traps 5.3.2; exit lists 5.3.3

1.9 Asynchronous Processing

An OS/2 system is host to a herd of independent processes, each representing a unique address space. The processes are home to flocks of threads that execute concurrently and asynchronously. There is a wealth of methods by which the processes may share and exchange data and by which the threads may coordinate their activities.

1.9.1 Time Slicing

OS/2 time-shares the CPU among all loaded programs. It keeps a list of all threads that are ready to run. Timer interrupts occur at regular intervals, and on each interrupt OS/2 gets control and makes a decision about which thread shall use the CPU next.

OS/2 uses a preemptive scheduling method; that is, it can take control away from one thread and give it to another (under the alternative, nonpreemptive scheduling, a thread would hold the CPU until it released it voluntarily). There are three dispatching classes and 32 priority levels within each. After each time-slice, OS/2 gives the system to the thread with the highest priority in the highest class; within a priority and class it dispatches threads by turns in what is called round-robin scheduling.

Usually, most threads operate at a single priority level within the middle class, so scheduling is usually round-robin. Most of the time there are only a few ready threads. When there is only one, the system can waste time taking repeated timer interrupts. In this case, the length of a time-slice is gradually increased up to a maximum (set at configuration) to reduce the scheduling overhead.

On the other hand, when many threads contend for the CPU it is possible for one to be locked out. A maximum-wait parameter (set at configuration time) is applied; when a thread is delayed longer than the maximum wait, its priority is boosted up a step so as to ensure it gets a time-slice.

References: threads 9.1; dispatching 9.1.5, 9.1.6

1.9.2 Shared Storage

Ordinarily the address space of one process is completely disjoint from that of another; the two don't overlap, so that the segments of one process are simply invisible to the other. However, there are ways in which processes can share segments in a controlled way.

Code segments loaded from an executable file are always read-only. These are shared between processes whenever possible. The code segments of a dynamic link package are held in common by every client process. Furthermore, OS/2 never loads a second copy of the code of an application; if the operator starts two (or more) instances of the program there will be no disk I/O done for the code of the second instance and the two will share the only copy of the code segments.

Data segments are usually private, but they can be shared between processes through several mechanisms. The data segments of a dynamic link package may be marked *single* during linking and will then be treated like code segments: a single copy will be shared by all client programs.

Data segments allocated by calls to the system may be shared by any of three mechanisms. A segment may be named; then any process in the system may have the segment inserted in its address space by calling the system with the character-string name of the segment. Or an allocated segment may be shared between two (or more) cooperating processes by two different, but related schemes called Giving and Getting.

> **References:** shared segments 2.4.6; linking data segments *single* 3.4.4.1; sharing allocated segments 7.4

1.9.3 Semaphores

When read-write storage is shared between concurrent, asynchronous threads (this can happen when the threads are in different processes that share a segment, or are in the same process sharing its private storage) there must be some way of coordinating access to shared objects. OS/2 meets this need with a flexible, efficient scheme of *semaphores.*

A semaphore is simply a doubleword (32-bit integer) in storage. Threads can synchronize their actions in two principle ways. They can use a semaphore as a key, representing the right to modify a particular object, or they can use a semaphore as a signal, on which they wait for news of a particular event. (Those

familiar with other operating systems might recognize these as Resource Control Blocks and Event Control Blocks, respectively.)

The semaphore services are efficient and flexible. They can be used by any set of threads that have access to doubleword of read-write storage in common, even when the threads are in different processes. Threads that share no address space may still coordinate using *system semaphores*, which are named, system-wide objects maintained by OS/2. A thread in any process can create a system semaphore and any other process can get access to it by passing the character name to a system call.

References: semaphores 9.4; system semaphores 9.4.5

1.9.4 Timer Services

The system maintains constantly-updated time values at three precisions: milliseconds since last bootstrap, seconds since 1/1/1970, and a conventional day-month-year timestamp. Read-only access to all these time values can be obtained by a system call.

Interval timer services are available as well. A thread can request suspension for a given interval. The system can be asked to clear (signal) a specified system semaphore after a given interval, or at regular intervals. All intervals are specified in milliseconds and are accurate to plus or minus one timer tick of 32 milliseconds.

References: time and date information 5.1.2; timers 9.5

1.9.5 Queues

A queue is a named, system-wide object that may be created by any process and opened by any others who present the name. Queues may be operated under FIFO, LIFO, or priority-ordered rules.

A thread in any process may write a four-word item into a queue, while threads in the creating process may inspect and remove items. The threads that read a queue may wait for the next item, or may request that a semaphore be posted on the next write.

Since queues are system-maintained, named objects they are convenient for communication between processes. However, queue items are only four words in size; to pass larger objects the processes must have data storage in common so that they can pass pointers in the queue.

References: queues 9.7

1.10 Video Display Control

OS/2 offers extensive services for control of the virtual display screen, and less convenient mechanisms for control of the physical display adapter.

1.10.1 Text-Mode Displays

The video displays attached to OS/2 systems operate in one of two mutually-exclusive modes, *text* or *graphics*. In text mode the screen hardware displays a grid of character cells. The contents of each cell is specified by a two-byte field containing a character byte and an attribute byte that specifies color and brightness.

A process does text-mode screen output by writing character bytes, attribute bytes, or both into a logical screen buffer, using one of a family of convenient system calls. Whenever the screen group switches to the foreground, the logical screen buffer is copied into the physical display buffer and the output becomes visible.

The system functions for text-mode output are fast and convenient, and are adequate to implement, for example, a full-screen editor with multiple windows.

References: stream output to display 10.2; text-mode output 10.4.2

1.10.2 Graphics-Mode Displays

The display hardware can usually be switched into a graphics mode, in which the screen contents are controlled pixel-by-pixel. Pixel values are specified by one of many incompatible data formats depending on the specific mode. In some modes display storage is multi-planed, so that the one address in the buffer has four different meanings depending on the setting of hardware registers in the display adapter.

The permutations of graphics modes were apparently too much to cope with, so OS/2 does not give direct support for graphics modes of the virtual display screen. Instead, it supplies a scheme by which multiple processes may share access to the physical display hardware. There are system calls by which a process can wait for its screen group to come to the foreground, and by which it can have the physical display buffer added to its address space.

There is also a mechanism by which a process can set up an asynchronous thread that is suspended until the process is about to be switched to the background; the watchdog thread may then save the display hardware status and display buffer contents. A second watchdog thread can be set up to restore the display status asynchronously when the process is switched back to the foreground.

References: graphics display output 10.5; watchdogs 10.5.4, 10.5.5

1.11 Keyboard Input

Ordinarily a program will read its primary input from the standard input handle it inherits. Interactive programs, however, require low-level access to the keyboard. This is available; a process may read the scan codes of individual keystrokes direct from its virtual keyboard, whether that is standard input or not.

It may also establish a new keyboard translation map (code page), and may feed generated scan codes through the translation process to see what keystrokes they translate into.

There is some provision for multiple processes to share a single virtual keyboard. The *keyboard focus* is the right to receive input from the virtual keyboard. A process may claim or release the keyboard focus. If two or more processes contend for input, the one with the focus will receive it.

References: keyboard input 11; keyboard focus 11.3.1

1.12 Mouse Input

A process may open the virtual mouse of its screen group for input. The process may specify or query the shape and position of the mouse pointer, and may read events (mouse motions and clicks) from the mouse event queue.

The mouse pointer is drawn automatically when the display is in a text mode. When the screen is in a graphics mode, mouse events can be read and will contain coordinates given in pixels, but the mouse pointer must be drawn by the application.

Reference: mouse input ch. 12

1.13 Printer Output

Printers are managed on a system-wide basis. Any process can open a file handle to the printer (under the name *PRN*, *LPT1*, etc.) and write. The printer support will accept output from the first file to be opened. Other processes may open files to the printer but will be blocked when they attempt to write, until the first process closes its file.

This describes the basic support. The system contains support that permits a much more flexible, disk-based print spooler subsystem to be built. A rudimentary print spooler is included with some versions of the system, and better ones will no doubt become available from third parties.

1.14 Device I/O and Device Monitors

Applications can in certain cases achieve detailed control over hardware devices.

1.14.1 Device Drivers

The actual control of hardware devices rests with programs called *device drivers*. Each of these specialized programs manages a device on behalf of the OS/2 kernel. Ordinarily, applications request operations of OS/2 in terms of file access. OS/2 translates application requests into a larger number of simpler hardware operations and calls on the device drivers to perform these operations.

The design and construction of device drivers is not covered in this book.

1.14.2 Generic IOCTL

Some device drivers support the Generic I/O Control Interface, or Generic IOCTL. There is an OS/2 system call, *DosDevIOCTL*, by which an application

may pass a request packet directly to a device driver. If the driver supports Generic IOCTL, it will respond to certain request packets.

The keyboard, mouse, and screen device drivers respond to Generic IOCTL. However, they will perform many of their IOCTL functions only for the very first process ever to request them. That is invariably the Session Manager. The remainder of their IOCTL functions are only duplicates of functions that are available as normal system calls.

The serial-port device driver supports the two standard IBM PC/AT serial ports, COM1 and COM2 (and COM3 in the PS/2 line). Input and output are done by file reads and writes, but status checking and control of the ports is done with IOCTL functions.

References: device drivers 13; IOCTL mechanism 13.4

1.14.3 Device Monitors

A device monitor is an application thread that is allowed to sit between the device driver and the operating system. A monitor sees every byte of the data stream going to or from a device. Furthermore, the monitor can censor the data stream, deleting data or inserting new data.

A monitor thread is registered with a system call that is available to any process. More than one monitor may be registered for a given device. The system organizes monitors into a pipeline—a bucket brigade—in which the output of one monitor becomes the device input of the next.

The system maintains a separate monitor chain for each virtual keyboard, and a single monitor chain for each printer device. Keyboard monitors can be used to implement keyboard macro systems, or can be used as a means of watching for a hot key on behalf of a pop-up application. A printer monitor is the foundation for building a print spooler.

Reference: device monitors ch. 16

1.15 System Extensions

The OS/2 system is designed from the ground up to make it easy to add new subsystems. A complete new functional subsystem may be added in the form of

a new dynamic link library and a few named queues and storage segments—and is immediately as accessible as every other feature of the system.

As a result, each vendor that relabels OS/2 is free to add proprietary features and probably will. IBM has announced plans for Extended Editions containing elaborate support for database and communications applications. These will be delivered as new dynamic link packages. The Presentation Manager, a windowing interface, will be (or at least in principle could be) delivered as new dynamic link libraries. There is every reason to expect third parties to market new subsystems for OS/2.

Then what is OS/2? It is really a family of overlapping operating systems. However, the functions described in this book are part of the core system and should be available in every OS/2.

2

The Hardware Environment

The design of OS/2, and the structure of OS/2 programs, is formed and limited by the hardware on which it runs—the Intel 80286 CPU, or 286 for short. This topic is a survey of the hardware of the 286. It is meant for programmers who, because their experience has been mostly with high level languages, have not learned the hardware the way that programmers who use assembly language do. It aims to cover the things that are fundamental to an understanding of:

- subroutine linkage conventions
- instruction patterns seen when using CodeView
- memory models
- the rules and methods of program linking
- dynamic linking
- how multiple threads share an address space

This is not an introduction to the whole instruction set of the 286. The best reference to that is the manufacturer's *Programmer's Reference Manual* (Intel order number 210498), especially when used in conjunction with one of the many good introductions to assembly-language programming.

Nor is this an introduction to the unique features of the 80386, the more powerful member of the Intel family. While OS/2 will run in a 386, it treats it as a 286 and doesn't support its unique features.

2.1 Fundamental Ideas

In order to follow this discussion you should be comfortable with these fundamental points:

- The concept of number systems other than decimal, in particular the *binary* number system and the *hexadecimal* system that lets us write binary numbers in compact form.
- The concept of the *bit*, the *byte*, and the 16-bit *word*.
- The important numbers relating to the byte and word: that a byte counts 256 things from zero to 255; that a word counts 65,536 things from zero to 65,535.
- The concept of how instructions and data alike are represented as binary numbers stored in bytes and words in computer memory.
- The concept of a stack.

If you don't feel at home with these ideas you will have difficulty with this topic (and probably with program linking and CodeView as well), and you should probably brush up on these and related points.

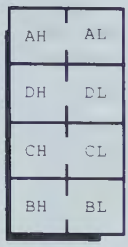
2.1.1 Depicting Computer Storage

A diagram like the one in Figure 2.1 can be baffling to the uninitiated. Here is the key. The fundamental unit of information is the *bit*, but the fundamental unit of storage is the *byte*, a group of eight bits. Neither a bit nor a byte really looks like anything in particular, any more than an electron or an average looks like something. However, programmers usually think of a byte in storage as a square box or pigeon-hole. Since a byte's-worth of bits can represent one printable character, most people depict a byte as a box of about the size needed to hold a capital letter.

The 286 also deals with storage in units of *words*. A word is 16 bits or two bytes, so most people depict a word as a box that's about twice as long as it is high. That is how words are drawn in this book.

The intent of Figure 2.1 should now be clearer. The machine registers are special storage locations built into the CPU. They are referred to by their two-letter names, *AX* and so forth. Most of them hold 16-bit words. Some, it appears, can be treated as adjacent bytes (square boxes) as well.

Figure 2.1
The Registers of the 80286



AX: general accumulator

DX: doubleword accumulator

CX: counts and loop control

BX: miscellaneous addressing

SI: string from-ptr and misc.

DI: string to-ptr and misc.

BP: stack-frame base

SP: stack pointer offset from SS

IP: instruction offset from CS

CS: code segment selector

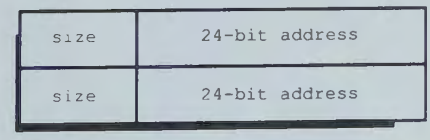
SS: stack segment selector

DS: data segment selector

ES: data segment selector

TSR: task state segment selector

LDTR: Local Descriptor Table selector



GDTR: Global Descriptor Table

IDTR: Interrupt Descriptor Table

2.2 The Machine Registers

The 286 CPU contains a number of *registers* (see Figure 2.1). If you are not used to dealing with the machine at this level it may help to realize that a register is simply a storage location, much like a program variable except that it is located within the CPU chip and is intimately connected to the rest of the CPU's circuitry. In fact, the registers are the global variables of the CPU—and the machine instructions are just small subroutines that take their parameters from, and leave their results in, these global variables.

To *load* a register is to assign a value into it from elsewhere in storage; to *store* a register is to assign the register's contents to a storage location.

The first thing to understand about the registers of the 286 is their irregularity. Every single register has a unique set of uses! This is unusual; most computer designs aim for regularity and uniformity among registers. Each register of the 286, however, must be learned as a special case. A serious programmer must learn them for two reasons: the design of the registers is fundamental to the crucial topic of how storage is addressed. And the registers are displayed by CodeView and it is sometimes important to understand how one's language compiler is employing them.

2.2.1 The Classes of Registers

The registers can be divided into clusters, as has been done in Figure 2.1. Those in the topmost group (*AX* through *BX*) are alike in that each can be used either as a word or as a pair of independent bytes. That is, a machine instruction may operate on a word under the name *CX*, or on the most significant byte of the word as *CH*, or on the least significant byte of the word as *CL*. The registers *AX* through *BX* are primarily used as operands of arithmetic and binary operations.

The registers of the second group (*SI* through *IP*) are alike in that they may only be operated on as word quantities, and in that all can be used to address storage. Some of them can be used for arithmetic as well, and *SI* and *DI* are often used by compilers to hold temporary results or for variables declared with the register attribute.

The registers of the third group (*CS* through *LDTR*) are alike in two ways. None may be used for any operation except loading and storing. All of them contain the *segment selectors* that are the fundamental storage addresses in the 286.

The bottom four registers, from the *TSR* through the *IDTR*, determine the contents of the *address space* of the current application program, that is, the totality of storage that it is able to address. An ordinary program is incapable of loading these registers; the OS/2 kernel alone can do that. However, any program may examine their contents.

Reference: address space 2.4

2.2.2 The Arithmetic Registers AX, DX, CX, and BX

The *AX* register is the most-used register. One reason is that there are a number of machine operations that work only on *AX* or its bytes *AH* and *AL*. Integer multiplication and division, for instance, work only on the contents of *AX* and its extension, *DX*. Decimal arithmetic can be done only in *AH* and *AL*.

By programmer's convention, moreover, *AX* is the register of choice for passing parameters and results between subroutines. A function that returns either a byte or a word result in C or Pascal, returns it in *AX*. All the OS/2 functions that return an error code return it in *AX*.

The hardware operation of multiplying two words creates a 32-bit result with the most significant 16 bits being deposited in *DX*. For this reason, *DX* is considered the extension of *AX*. In some language implementations, a function that returns a 32-bit result (a long integer or a normal float) does so in *DX* and *AX* (the more significant half in *DX*).

The *CX* register is used as a count-down register in several machine instructions. The *LOOP* instruction, for instance, decrements *CX* and jumps to the top of a loop if it remains nonzero. The string instructions that operate on variable numbers of bytes or words take their count out of *CX*, as do some shift and rotate instructions.

The *BX* register is the only one that can be treated as a pair of byte registers and can also be used to form an address of a storage operand.

2.2.3 The Pointer Registers SI, DI, SP, BP, and IP

Together with *BX*, the pointer registers (*SI* through *IP* in the figure) are the only ones that may be used to form the addresses of operands in storage. We will say more about how addresses are formed later. For now it is enough to say that the

address of a location in storage is a binary number. The CPU forms the address of an operand by adding the contents of registers, and the numbers in the pointer registers may be used this way. They can also be used as operands in most arithmetic and binary instructions.

Three of these registers are fully dedicated to special uses. The two that are not, *SI* and *DI*, have special uses in the string instructions but are otherwise usually employed as miscellaneous word registers.

The Instruction Pointer, *IP*, is always used to address the next instruction to be executed. It scans through storage as instructions are fetched and processed. Jump instructions modify it, causing the program to jump to some instruction other than the next one in sequence. Call instructions push its value onto a stack and then modify it.

The Stack Pointer, *SP*, is always used to address the last word pushed onto the machine stack.

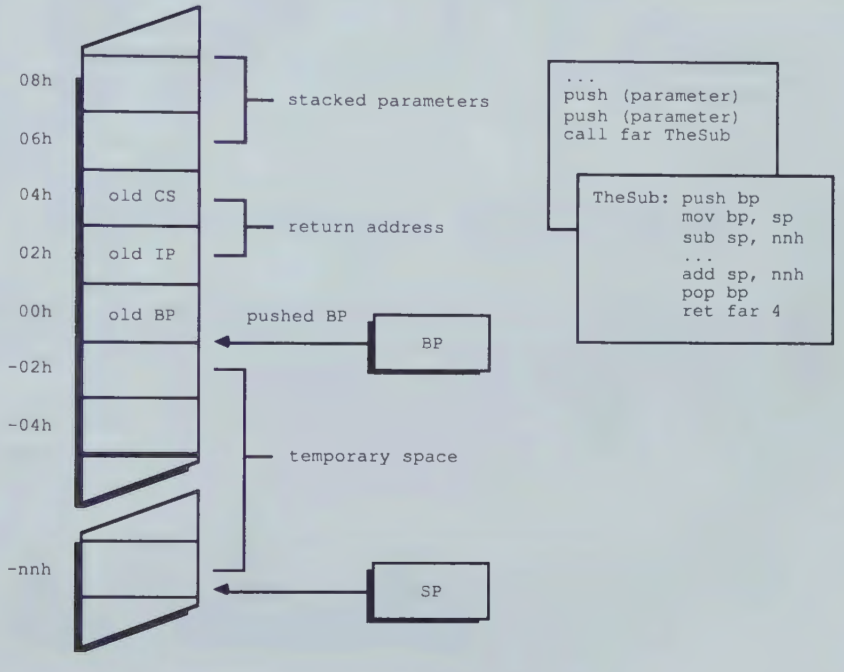
To *push* is to decrement *SP* by two and then to store into the word to which it points. There are instructions to push registers or other values onto the stack, and the Call instructions push the return address (*IP*) onto it. Since *SP* is decremented on a push, the stack grows from higher-numbered locations toward lower-numbered ones. Most people think of the stack as growing downward like a stalactite from the ceiling of a cave. Decrementing precedes storing, so *SP* always points to the last word pushed, not to the place the next push will go.

To *pop* is to load the word that *SP* points to and then to increment *SP* by two. The Return instructions pop a pushed return address from the stack into the *IP*, thus effecting a return from a subroutine.

2.2.3.1 Subroutine Calling

The Base Pointer, *BP*, might better be called the Frame Pointer, for that's how most compilers use it. The universal convention for calling from one routine to another is illustrated in Figure 2.2. You will see similar instruction sequences over and over when you examine compiled code with CodeView. This is also the method by which programs call OS/2 functions.

Figure 2.2
The Conventional Stack Frame



The caller places parameters on the stack with push instructions. The far call instruction pushes a 32-bit return address on the stack and sets the CPU to execute the first instruction of the called procedure.

Typically that first instruction will be *push bp*, to save the value that the calling procedure had in *BP*. The second will be *mov bp, sp*, that is, copy the contents of *SP* into *BP*. Since *SP* pointed to the last-pushed word, *BP* now does also. Since *BP* may be used in forming addresses of operands, the parameter values may now be addressed by fixed displacements from *BP: BP+6* is the offset of the last-pushed parameter word.

Most routines need space for variables that are local to the routine. Space is acquired by deducting the number of bytes needed from *SP*. That has the effect of reserving a block of bytes on the stack. The words and bytes within this space may be addressed at negative displacements from *BP*; the first one is at *BP–2*. Thus *BP* anchors the called routine's *stack frame* of local variables and parameters. The space is easily released at the end of the routine by adding its size to *SP*.

2.2.3.2 Parameter Order Conventions

There are two incompatible conventions for pushing a subroutine's parameters onto the stack. The question is, which is to be the first parameter: the one that is first pushed onto the stack (and thus the one at the greatest offset from *BP*), or the one that is last pushed onto the stack (and thus the one nearest to *BP*, lowest in the stack frame)?

The question doesn't arise in assembly language programs, but it does in high level languages. Consider the call *doit(x,y,z)* where the three parameters are integers (words). How will they be laid out on the stack?

A Pascal compiler will generate push instructions from left to right, so that the leftmost parameter in the list will be pushed first and the rightmost, last. Under CodeView, the Pascal code for *doit(x,y,z)* will disassemble to

```
push x
push y
push z
call doit
```

This very reasonable method only becomes awkward when the number of parameters is not known in advance. That never happens in Pascal or Modula, where a routine's parameter list must be declared in full. The C language, however, permits functions to have varying numbers of parameters. For this reason most C compilers generate parameter pushes from right to left, pushing the rightmost parameter first and the leftmost, last. Under CodeView, a call of the C function *doit(x,y,z)* will disassemble to:

```
push z
push y
push x
call _doit
```

(By convention, C prefixes the names of all user-defined subroutines with an underscore.)

This code sequence puts the first-written parameter lowest on the stack, at a known offset from the frame pointer. The first-written parameter is usually the one that will tell the called routine how many more parameters are coming. For example, the character template that is the first parameter passed to *printf()* tells it how many parameters to convert and display. The Pascal convention would leave

this parameter at an unknown offset into the frame, while the C convention leaves it at *BP+6* regardless of how many parameters are passed.

It's because of this conflict that the reference pages of this book show the parameter lists of OS/2 functions as graphic depictions of the stack, not as function calls in some language.

Reference: reference pages begin on page 467

2.2.3.3 Subroutine Return Conventions

There are two conflicting conventions for clearing parameters from the stack when a subroutine is finished. In one convention it is the subroutine's responsibility to discard stacked parameters. This can be done by giving a numeric operand to the Return instruction. That instruction pops the return address from the stack into the machine registers, effecting a return to the next instruction after the call. When given an operand, it adds it to *SP* after the pop. That effectively discards that many bytes of stacked parameters.

This convention is the most economical of code space, since the stack clearing is done in just one place, the subroutine, and not after every call to the subroutine. Most assembly language programs and all Pascal programs use this convention. So do the OS/2 system calls.

But the C language permits functions to have a varying number of parameters. It's awkward for the subroutine to discard a variable number of parameters from the stack, whereas the calling code knows exactly how many words it pushed. Therefore most C compilers require the calling code to do the clearing. The called function merely returns, and the caller does an *add sp,n* to discard the *n* bytes of parameters it stacked.

Obviously these two conventions can't be mixed. If they were, either no parameters would be cleared or too many would be. Either way, the stack would be disordered and the program would crash.

The OS/2 C compilers support a Pascal keyword for the declaration of external functions that use the Pascal conventions for stack parameters. Since the OS/2 system functions use that convention, they must be declared in C with the Pascal keyword, as follows:

```
extern void far pascal DOSBEEP(int);
```

The C compiler will realize that *DOSBEEP* will discard the parameter.

2.2.4 The Segment Selectors

Storage is addressed in segments. A *segment* is a sequence of adjacent bytes—from one to 65,536 of them. The position of a byte within its segment is its *offset*. The offset of the first byte in a segment is zero; the second is at 1, and so on to the last byte, whose offset may be as large as 65,535—or 0ffffh, the largest number that can be recorded in a 16-bit word.

For every addressable segment there is a number that is the *selector* of that segment. A selector is held in a 16-bit word. The full address of any byte includes both its selector number and its offset. Such a complete address is called a *far pointer*; it fills 32 bits of storage (two adjacent words).

An offset alone is sometimes called a *near pointer*. You must understand that an offset alone is useless. It's like a house number without a street, or a first name without a last. The segment to which an offset applies must always be known, either explicitly or implicitly. A near pointer may be stored, but when it is used it must be interpreted with respect to some implied selector.

No storage may be addressed as an operand of a machine instruction until its selector has been loaded into one of the selector registers: *CS*, *SS*, *DS*, or *ES* (*TSR* and *LDTR* are special cases). That's worth emphasizing. The one point of consistency in the architecture of the 80286 is that all storage references, without exception, require the use of a segment register. If the selector for a byte isn't in a register, the machine cannot touch it, and that is that.

2.2.4.1 The Stack Selector

The Stack Selector register, *SS*, always contains the selector for the segment that contains the machine stack space. Thus *SS* and *SP* together form the far pointer to the word last pushed onto the stack.

The offset in *SP* is always interpreted with respect to the segment named by *SS*. The offset in *BP* usually refers to that segment as well, since *BP* usually has the offset to a stack frame.

While the stack is vitally important, it is not usually large—typically from one to a few kilobytes. The segment that contains the stack—the segment whose selector must be in *SS*—could be devoted to the stack, but usually it is not. It is typically a segment of miscellaneous data, of which the space devoted to stack is only one portion.

This is a convention of the compilers and linkers, and not altogether a good one. When the stack is only part of a larger segment, there is no hardware protection against stack overflow or underflow. That is, there's nothing to stop *SP* from decrementing down into the data below the space intended for stack (overflow), or from incrementing until data is popped that was never stacked (underflow). If the stack segment were purely for that purpose, the CPU hardware would detect either of these mistakes and stop the program with a segmentation error. (It would even be feasible for the operating system to recognize stack overflow, enlarge the stack, and restart the program. However, OS/2 does not offer that support.)

The reason that the Microsoft compilers (at least) mix stack and data space is that the same compilers generate code for real-mode MS-DOS programs. In DOS nothing is lost to the practice, because there is no hardware limit on any segments, stack or otherwise. As compiler writers understand the protect-mode environment of OS/2 better we may see more stack-only segments used.

2.2.4.2 The Code Selector

The Code Selector register, *CS*, always designates the segment of code that the machine is executing. As noted earlier, the offset of the next instruction is always in *IP*, so *CS* and *IP* together are the far address of the next instruction the machine will do.

2.2.4.2.1 "Far" and "Near" Calls and Jumps

A *far jump* is a jump that loads both *CS* and *IP*; it's a jump to some offset in a different segment of code than the current one. A *far call* is a call that pushes both *CS* and *IP* on the stack and then loads both; it's a call from code in one segment to code in another.

A *near jump* is a jump that only loads a new offset into *IP*; it's a jump to another instruction in the current code segment. A *near call* is a call that only pushes *IP* on the stack and loads a new offset into it; it's a call to a routine in the current code segment. Generally speaking a compiler will translate one source file into

one code segment, so that calls to routines in the same source file will be near calls, and calls to external routines will be far calls.

Clearly if there are two kinds of calls, there must be two kinds of returns: a near one that only pops *IP*, and a far one that pops both *IP* and *CS*. Furthermore, the kind of return with which a subroutine ends dictates the kind of call that must be used to invoke it. Picture what would happen if a subroutine that ended in a far return were called with a near call: the near call would push only *IP* while the return would pop *IP* and then *CS*, so a word would pop into *CS* that wasn't pushed by the call, a word that probably wasn't a selector for a code segment at all. In the opposite case—a near return to a far call—an offset would be popped into *IP* but *CS* would not be loaded. An offset apart from its selector is irrelevant; in this case, control would pass to an irrelevant spot in the subroutine's segment.

In assembly language all procedures and public labels have to be designated as near or far, so the assembler is able to generate the correct kind of call and return instructions automatically.

In Pascal, routines that are public are compiled with a far return. All first-level routines in a module or unit are public, as is any routine declared with the public attribute. Other kinds of procedures and functions are assumed to be near procedures and are compiled with a near return. Calls to external routines and to routines declared with the public attribute are compiled as far calls, while calls to nonpublic routines (which must be in the same source file) are compiled as near calls. Since the OS/2 system calls are declared external, Pascal generates far calls to them.

In Microsoft C, functions may be declared with explicit near or far keywords. A function declared as *far* is compiled with a far return. A call to a function declared *extern far* is compiled as a far call.

When these keywords are omitted, the C compiler assumes different things depending on the memory model it is using (more on memory models shortly). In the Small model, all code is assumed to be linked into a single code segment, so all calls and returns will default to near ones. Under the other models, external functions are always assumed to be far. Therefore functions that are publicly accessible (ones not declared static) are compiled with far returns since they can in principle be declared external in other source files.

In C, the OS/2 system functions should be declared as *extern far* so that the compiler will generate a far call even in a program compiled to the Small model.

2.2.4.3 The Data Selectors

There are only two registers with which to name segments of data, that is, the segments that contain space for variables and arrays. The common one is the Data Segment register, *DS*. Then, as an afterthought and not as if anyone might really need it, the CPU provides an Extra segment register, *ES*.

A program may refer to constants that are part of the program text using *CS*. And it may address stacked parameters and local variables in the stack frame, using *SS*. But any other data item must be in a segment whose selector is in either *DS* or *ES*. If the program needs to refer to a data item in a different segment, that segment's selector must be loaded into *DS* or *ES*, replacing whatever selector was there.

As a result programs are full of loads into *DS* and *ES*. Now, I must confess that it makes me uneasy when I think of loading *DS*. My subsconcious, it seems, knows that *DS* has to be loaded from some variable in storage, and that most program variables are already based by *DS*. Therefore (my subconscious frets) once *DS* is loaded with some other selector, we will never get back to the segment with the variables that we need to load it from. (My subsconcious has been programming for a long time.)

The fact is that the source of most loads into *DS* and *ES* are immediate operands of instructions: constants that are embedded in the instructions that use them. Strangely, there is no instruction that will load a constant directly into *DS* or *ES*; only instructions to load the segment register from another register. Thus a sequence like this is often seen under CodeView:

```
mov  ax, some binary value
mov  ds, ax
```

If it's an assembly language program, the source probably reads like this:

```
mov  ax, SEG SomeVar
mov  ds, ax
assume ds:SomeVar
```

The instruction that loads *AX* contains, as part of the instruction text, a 16-bit number that is the selector of the desired segment. In the topic on linking we consider how that constant is created, how it may be modified during linking, and how it is finalized when the program is loaded.

There are other ways of loading data selectors: from the stack, for one. Compiler-generated code often wants to make *DS* and *ES* name the same segment, and will do it with:

```
push ds
pop  es
```

or the reverse.

It is also common for a subroutine parameter to be a far pointer—a two-word address containing an offset and a selector. There are machine instructions for loading far pointers and they are often used by compiler code to pick up the far address of a parameter with an instruction like:

```
les  di, bp+n
```

The far pointer is at offset *n* in the stack frame. The offset it contains is loaded into the specified pointer register (*DI* in the example), and the selector from it goes into *ES*. There is a similar instruction, *LDS*, that loads a pointer and *DS*. The far pointer may be a parameter in the stack frame or a pointer variable in a data segment.

> **References:** selector constants in the object module 3.2.1.3; resolving linkages 3.2.2; fixing up references at load time 4.1.2.3

2.3 The "Memory Model"

The 80286 has only four segment selectors that can be used to address data. At any instant the program can refer to constant data in the code segment, to temporary data and parameters on the stack, and to variables in two data segments—and that's all. To further restrict the program, the farthest it can reach into any of these segments is the 64K bytes that can be coded in a 16-bit offset. Before it can refer to any other segment it must load a different selector. But into which register? *CS* and *SS* can't be changed casually or the program will lose its stack or its very code.

The program must employ this limited selection of registers in such a way as to address all the code space, stack space and data space it needs while doing as few loads of segment registers as possible. The method that the program uses, its general strategy for arranging data into segments for greatest convenience and efficiency, is called its *memory model*. An assembly language programmer is

free to devise whatever strategy suits the program best. However, a few strategies are conventional and most programs use one of them. In particular, the Microsoft language compilers always use one of the conventional models.

2.3.1 The Tiny (One-Segment) Model

The simplest strategy is to declare that the entire program, code and data alike, must fit into a single segment. Then all the segment registers may be loaded with the identical selector. After that, they never need to be loaded again. All calls and jumps are near ones, since all the code is in the one segment. No far pointers are needed for data; what's more, the program doesn't need to distinguish between near pointers that are relative to the stack, to the code, or to the data segments. Any offset may be used with respect to any segment register.

Since many simple programs fit very comfortably into a block of 64K bytes, the Tiny model could be useful. MS-DOS supports it, but requires that programs that use it must be stored in files with the .COM file extension. When MS-DOS loads such programs it sets up their segment registers with identical selectors and sets the *SP* register to the end of the program segment, so the stack grows down from the top of the segment.

OS/2 requires code segments to be pure, read-only segments. Therefore it cannot permit program variables and program code to be mixed in a single segment, and therefore OS/2 does not support the Tiny model. Tiny-model programs can be constructed with the OS/2 assembler, and the OS/2 Link command in real mode will link them, but they cannot be loaded and executed under OS/2.

2.3.2 The Small (Two-Segment) Model

The next simplest strategy is to put all the program code in one segment and all program variables and the stack into another. The selector for the code segment is loaded into *CS*, while *SS*, *DS*, and *ES* are identically loaded with the selector for the one data segment.

In this Small model all calls and jumps are near ones. Any offset to data may be used indifferently with respect to *SS*, *DS*, or *ES*. However, if the program wanted to store a pointer to constant data located in the code segment, that pointer would have to be a far one. As a result the use of the Small model usually implies that constants are kept in the data segment.

The Small model is the most common model for assembly language programs, at least those that are written as a single source file. Of the Microsoft languages, only the C compiler supports the Small model.

References: near and far pointers 2.2.4; near and far calls 2.2.4.2.1

2.3.3 The Medium Model

The Medium model is the one most language compilers use. It allows the code of a program to occupy as many segments as necessary, but it assumes a single data segment to which both *SS* and *DS* point.

The Medium model comes about very naturally from the design of most programming languages. Each source module becomes a segment of code. Calls and jumps within the segment are compiled as near calls and jumps. Calls to labels external to the source module are compiled as far calls.

The Medium model is constrained by the single data segment it permits. A program's constants, global variables, stack, and heap must add up to no more than 64K bytes. This feels very cramped today, but it has benefits for the designer of a compiler. When there is only one data segment (and provided constants are located in the data segment not in the code segments), all pointers to data can be near pointers. The important outcome of this is that pointers to stacked data do not have to be handled any differently from pointers to variables in the heap, to global variables, or to constants.

When all data pointers are alike, parameters passed by reference can be near pointers. In Pascal for example, *var* parameters are implemented by passing pointers to the actual parameters. A Pascal procedure declared with *var X:real* as a parameter will be called with a pointer to a real variable. The compiled procedure must work correctly regardless of whether its actual parameter is a global variable, or a local variable of the procedure that called it, or allocated in the heap. The only way to achieve this is to make sure that pointers to data are uniform—either all near or all far. Near pointers take less storage than far ones and the machine code to use them is simpler and faster. That makes the Medium model popular with compiler writers.

The run-time library distributed with Microsoft languages had its origins in the time when only the Medium model was supported. Buried in it are two assumptions about the Medium model that can cause trouble in some OS/2 applications. The assumptions are, first, that *SS* and *DS* contain the same selector and therefore

if *DS* has to be altered it can be restored by copying *SS* into it, and second that there is only one data segment and that *DS* always points to it. Code that makes these assumptions will work when it is linked directly with other Medium-model code, but will fail when linked, for example, as part of a dynamic link library.

References: near and far pointers 2.2.4; near and far calls 2.2.4.2.1

2.3.4 The Large Model

The next step is to allow data objects to fill more than 64K. As soon as one variable might have a different segment address from another one, a program must use far pointers for all parameters that are passed by reference. But there must also be a policy for placing data objects into segments. There will be more than one segment of data; very well, which variables go in which segment?

When a compiler creates multiple code segments, it generally makes one for each source file. The Microsoft C compiler supports the Large model, and uses that policy for data segments. While compiling one source module, it makes a data segment to hold the global variables declared in that source file.

2.3.5 The Huge Model

We generally assume that a segment cannot be larger than 64K bytes. This is not a limitation of the underlying storage; there is nothing to prevent a program from reserving a block of a million bytes. The limitation is in the addressing hardware, as we've seen: 65,535 is the value of the largest 16-bit offset. When people conventionally say that the 80286 only supports 64K-byte segments, what they really mean is that the machine can only *address* 64K bytes based on a one *selector*. A segment may be larger, but in order to address it the program must use more than one selector.

A program can operate on a single object larger than 64K. Two things are required. First, the program must have a selector for each 64K-byte chunk of the object in sequence. Then the program may assume that the second selector, with an offset of zero, gives the address of the byte that follows the one addressed by the first selector with an offset of 65,535. Second, a single element of the object—an integer, for instance, or any other unit that the program might try to copy or write to a file—must not span the joint between chunks. If that restriction is observed, the chunks don't even have to be contiguous in storage, nor does

there need to be any special arithmetic relationship between their selector numbers.

A Huge memory model is made by extending the Large model to permit single global variables to be larger than 64K bytes. These objects are called huge in both the OS/2 manuals and in Microsoft C. The C compiler makes a huge object into a series of data segments, all but the last one being 65,536 bytes long. The Linker will usually (but doesn't have to) keep these adjacent and in sequence.

OS/2 itself supports the dynamic allocation of data spaces greater than 64K bytes. When a program allocates a huge segment of data, OS/2 creates a contiguous expanse of data and returns a set of selectors that have a fixed arithmetic relationship so that any of them can be calculated from the first one.

References: allocating huge segments 7.3.3

2.4 The Address Space

At its most fundamental, computer memory is not divided into segments but is simply a huge array of homogenous bytes. OS/2 requires a megabyte and usually has two megabytes or more. All those millions of bytes are just alike, but each has a unique number, starting with byte zero and running up to the end. The *address space* of a program is that portion of storage that the program can reach.

2.4.1 The Real-Mode Address Space

When the 80286 CPU starts up it operates almost identically to its predecessor the 8086. This initial mode is called Real mode. The CPU remains in Real mode until a special instruction changes the mode. MS-DOS never executes that instruction, so an MS-DOS system always operates in Real mode.

In Real mode the CPU forms storage addresses in a simple and direct way. Recall that an address is always made from the contents of a segment register and an offset. In Real mode, an address is always just

```
(segment*16) + offset
```

The binary number in the segment register is multiplied by 16 (that is, shifted left four bits) and added to the offset. The result has 20 bits, enough to count to 1,048,575, so that is the maximum usable storage size while the machine is in

Real mode. (*The experienced reader will immediately notice that it is possible to construct an address with 21 significant bits. If the selector and offset are both 0ffffh, the result of the expression shown is 10ffefh. But this is hardware, not mathematics; the top bit is truncated and the address "wraps around" to 0ffef in low storage.*)

In systems compatible with the IBM PC/AT, four hundred kilobytes of this space are dedicated to the screen hardware and various ROM chips, yielding the well-known and often-cursed limit of 640K bytes of usable storage in an MS-DOS system.

In Real mode there is a simple arithmetic relationship between a byte's address and its selector: the selector is simply the most-significant 16 bits of the address, and the offset is the least-significant four bits. MS-DOS programs often rely on being able to form the addresses of significant BIOS and DOS variables this way. For instance, using

```
mov ax, 0b800h
mov ds, ax ; address screen RAM
```

in an assembly language program, or *DEF SEG &H40* in a BASIC program to address the DOS information area at 0400h.

In fact, in Real mode there is a whole family of selector numbers for any byte, each made by subtracting 1 from the prior selector and adding 16 to the prior offset. Some programs rely on this, constructing selectors by arithmetic on other selectors.

Finally, in Real mode all storage is alike and may be used in any way that the program pleases: for code, for read-only data, or for writable data. Any program may write, execute, or set its stack wherever it pleases. Some programs rely on this ability, but more often programs are the victims of it as they step on each other's data or are stepped on.

Reference: real mode 1.3.1

2.4.2 The Protect-Mode Address Space

The alternative to Real mode is called Protect mode. It's a profoundly different mode of operation that affects almost everything the CPU does, but mainly it is a change in the way the hardware treats selectors. In Protect mode the arithmetic

relation between a selector and an address is broken. A level of indirection is inserted between the value of the selector and the address it designates: a selector is not an address but an *index into a table of segments*. The table entry contains the address of storage, and also contains a limit on the size of the segment and a usage type.

In Real mode, a program can form any address by loading a constant into a segment register. In Protect mode a constant address like 0b800h (the screen buffer selector, in Real mode) is just an offset into a table; if the table isn't that large, or if that table entry is invalid, the program will be terminated. If the table entry exists but doesn't point to the screen buffer, the selector won't do what the program expects.

In Real mode, if *n* is a valid selector, then *n*+1 is the selector for the 16th byte of the same segment. In Protect mode, if *n* is a valid selector, *n*+1 may or may not be valid—and if it is, the address of the segment it selects has *no relation whatever* to the address selected by *n*.

In Real mode a program can address anything in the first Megabyte of storage. In Protect mode, a program can address anything that has been entered into its table of descriptors, which could potentially amount to many, many megabytes of storage. However, the program can *not* address anything that has *not* been entered into its table. The table is built by the operating system, so the operating system can keep the address spaces of different programs completely separate, and can keep critical system data out of the reach of all programs.

In Real mode a program can use a byte any way it pleases. In Protect mode the type of a segment is described in its table entry, and the hardware enforces the type. Code segments may only be executed and Data segments can't be executed at all; read-only segments may only be read, and so forth. A great number of program bugs that would be very hard to catch in MS-DOS are trapped and diagnosed instantly in Protect mode.

Now let's see how this powerful scheme works in detail.

2.4.3 Segment Descriptors

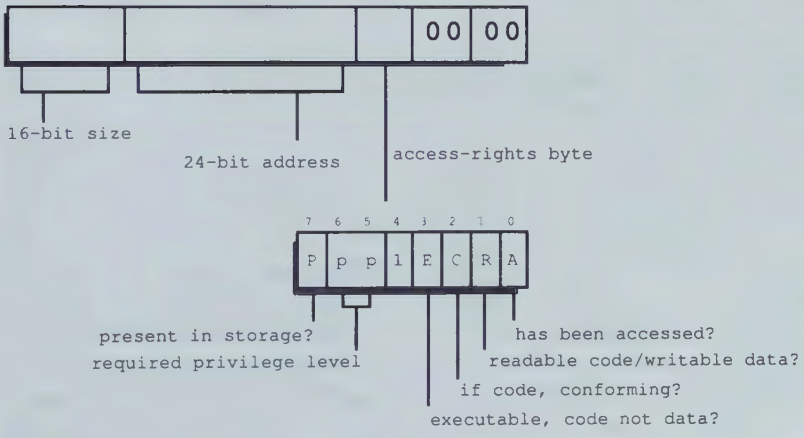
The segments of storage that a program may address are first allocated and prepared by the OS/2 kernel. Each one is carved out of the pool of space in the whole system and initialized with data from disk (if necessary); then it is entered into the table that will define the address space of the program.

The table entry is called a *segment descriptor*. It is a 64-bit data structure that describes the segment to the hardware. It is depicted in Figure 2.3. It has three important fields: a size, an address, and a byte of flags.

2.4.3.1 The Size of a Segment

The size of a segment is established by a 16-bit number in the segment descriptor. Effectively this is the largest offset that may validly be used with respect to the segment. The CPU hardware will not permit a larger offset to be accessed. If the program tries, the CPU will trap out (enter an interrupt handler) with a segmentation error and OS/2 will end the program.

Figure 2.3
The Form of a Segment Descriptor



In Real mode one may make a rule that a segment is thus-and-so long, but it will be only a moral stricture; there's nothing to stop a program from reaching beyond the limit. In Protect mode the program is literally *incapable* of that kind of mischief. It is this hardware feature that makes it desirable to have a dedicated stack segment. Stack overflow or underflow would lead to an offset in *SP* that violated the size field of the descriptor of the stack segment.

2.4.3.2 The Address of a Segment

The descriptor contains a 24-bit address of the first byte in the segment. Twenty-four bits can count up to 16,777,216, so that is the maximum memory size of an OS/2 system. When the program refers to a byte at some offset in the segment, the hardware actually adds the given offset to this address, and the result is the 24-bit address of the desired byte.

The segment's address is chosen by the OS/2 kernel. The program has nothing to say about it, and has no way to find out what it is. This gives OS/2 immense power to place segments to suit itself. It can move them around while the program is running (if the configuration file contains *memman=move*) and the program won't notice, and this means that it can compact storage when it needs to.

Reference: segment moving 7.1

2.4.3.3 The Access Rights Byte

A single byte in the descriptor determines the kind of segment and its permitted uses.

2.4.3.3.1 The Present and Accessed Bits

The Present bit tells the hardware whether the segment is actually in storage. If the bit is 1, the segment is located where the descriptor address field says it is, and may be used normally.

If OS/2 sets the Present bit to zero, the hardware will trap to an OS/2 kernel routine the instant the program tries to use this segment. This is the mechanism that allows OS/2 to swap segments to and from disk (when the configuration file contains *memman=swap*). When it needs more space, it finds a segment that it judges won't be needed soon. If the segment can't already be recreated from disk (a code segment from a loaded program could be recreated by loading it from the program file) the kernel writes the segment to the swap file. Then it can turn off the Present bit and reuse the space the segment had occupied. The space may be used to benefit the program that owned the swapped segment, or to make a segment for a completely different program.

When the program attempts to use its swapped-out segment, the hardware will trap to OS/2. The OS/2 kernel will recreate the segment in storage by reading it

from disk, then restart the interrupted program. The recreated segment might be in a different location in storage, but the program can't tell.

The Accessed bit is set to 1 by the hardware whenever the segment is used. OS/2 can use this as a hint as to which segments are actually being used and therefore should be swapped out last. Periodically it surveys all segment descriptors, noting the ones that have been accessed and clearing their Accessed bits to zero.

References: segment moving 7.1; segment swapping 7.2

2.4.3.3.2 The Code/Data Bit

One bit of the Access Rights byte is 1 if the segment contains code (machine instructions), and zero if it contains data. The CPU hardware will not execute instructions from a data segment. In fact, it won't permit the *CS* register to contain a selector for a segment whose Code bit is not 1. It will trap out to OS/2 which will terminate the program.

Ordinarily this is useful protection (how often have we seen a program take a wild branch and try to execute a data buffer?) but there are times when it gets in the way. Some programs create machine instructions on the fly and then need to execute them. For this rare use, OS/2 offers the *DosCreateCSAlias* service. It coins a code-type descriptor for a segment that is already known as data; then the program can use the same segment different ways with different selectors.

The hardware may or may not allow a code segment to be used for data. Another bit, the Readable/Read-Write bit specifies whether a code segment may be used as read-only data. If it's zero in a code segment, the hardware won't allow a selector for the segment to be loaded into any segment register but *CS*. When the Code bit is zero the segment is data; it can't be executed but it can be the source of operands for instructions. Then the Readable/Read-Write bit determines if the segment can be written to. If it is 1, the program may store into the segment. If it is zero, the segment is for input only.

The Readable/Read-Write bit can be established for any segment of a program when the program is linked.

References: *DosCreateCSAlias* 6.4; link options for read-only data 3.3.4.5.2; link options for execute-only code 3.3.4.5.3

2.4.3.3.3 The Privilege Level

Two bits in the Access Rights byte specify the *privilege level* of the segment. The CPU recognizes four privilege levels, from zero to three, with zero the proudest and three the most humble. There is a *current privilege level* stored in the CPU (discussed below). The CPU will not allow access to data in a segment whose privilege is prouder than the current privilege level. Nor will it permit jumps or calls into a segment that has a different privilege level except in very carefully defined ways.

Application programs and their segments of code and data are usually set to the humble level three. Segments that belong to OS/2 are set to snooty level zero. There is simply no way that an application program can refer to data in them or branch to code in them.

2.4.4 Descriptor Tables

The segment descriptors are organized into tables. The content of these tables determine what's addressable at any time.

2.4.4.1 The Global Descriptor Table

There is one *Global Descriptor Table* (GDT) that holds the descriptors for the segments that are common to all processes in the system. These are principally the code segments of the OS/2 system code and the data structures it uses for system management.

Look back to Figure 2.1. One of the registers built into the CPU is the Global Descriptor Table Register, the five-byte *GDTR*. It contains the size and the absolute address of the GDT. There is a machine instruction to load it; for obvious reasons it's privileged and only the OS/2 kernel can use it. But there's an instruction to store it, and any program may use that, so if you want to examine the *GDTR* you may store it into a variable and do so.

The GDT contains as many as 8,192 descriptors, each of the form shown in Figure 2.3. Only 8,191 of them can be used, though; the zeroth entry in the GDT is reserved for a purpose that will be described in a moment.

2.4.4.2 Local Descriptor Tables

There is at least one *Local Descriptor Table* (LDT). An LDT is a table containing as many as 8,192 descriptors, each of the form shown in Figure 2.3.

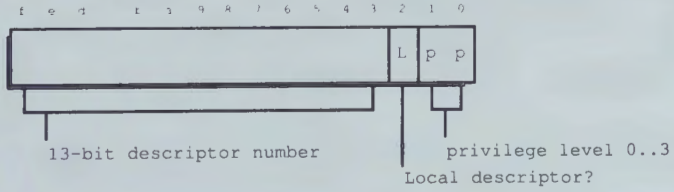
One LDT describes the segments that are available to *one process*. Each OS/2 process has its own LDT, and that LDT defines what the process can access. The segments it may use are in its LDT; any other segments (besides those in the GDT) are inaccessible to it.

2.4.5 Selectors and Addressing

Now we can see how addressing works. A *selector*, the value that a program loads into a segment register, is an index into a descriptor table. The selector selects the descriptor, and the descriptor tells the CPU what it needs to know about the segment, especially its address and size.

There is a little more to a selector. Figure 2.4 shows its complete contents. The most significant 13 bits are the index into a descriptor table. A 13-bit number can count from 0 to 8,191, which accounts for the maximum sizes of the GDT and LDT.

Figure 2.4
The Form of a Selector (Segment Number)



If bit two of a selector word is zero, the selector applies to the GDT. The all-zero selector word would select the zeroth entry in the GDT. However, the all-zero selector has been set aside as the null pointer value. It causes a hardware trap anytime it is used. If zero is set in any uninitialized pointer variable, an attempt to use an uninitialized pointer will cause a hardware trap. This sets a hardware watchdog on a very common and very troublesome kind of software bug.

When bit two of a selector word is 1, the selector applies to the LDT. But to which LDT, since there is one per process?

A descriptor table clearly must be contained in some segment of storage. The segment that contains the GDT is described by the address and length in the *GDTR*. Each LDT is contained in its own segment. To be usable an LDT must be in a global segment, that is, a segment that is described by an entry in the GDT.

Look back at Figure 2.1 and note the Local Descriptor Table Register, *LDTR*. It contains the selector for the current LDT. The selector will be a global one, that is, one whose bit two is zero. The hardware can index the GDT by that number to find the address and length of the current LDT.

There are machine instructions to load and store *LDTR*. The instruction to load it is privileged; only the OS/2 kernel may use it. When that instruction is executed, the CPU hardware loads *LDTR* and then immediately looks up the descriptor for the LDT and holds its address and size in the CPU circuitry for quick reference.

Before OS/2 dispatches a process it loads *LDTR* with the selector for that process's LDT. While the CPU is running in the code of the process, all local selectors will be referred to the LDT for the process. The LDT contains descriptors for every segment that the process may access. Those descriptors are readily addressable, while all other segments in the system are utterly, completely out of reach.

2.4.5.1 Loading a Data Selector

Whenever a program loads a selector, the CPU hardware goes through an involved process of loading and validation. Consider the very common sequence

```
mov ax, seg dgroup
mov ds, ax
```

which loads the selector for a program's *DGROUP*, its segment of miscellaneous static variables, into *DS*.

The CPU hardware doesn't know about *DGROUP*; it sees a binary selector value. Bit two will be 1, since *DGROUP* is local to a program and its descriptor will be in the LDT.

The CPU has known the address and size of the LDT since the program was dispatched. Now it checks that the selector does not index past the end of the LDT. If it did the CPU would trap out to OS/2.

The CPU indexes the LDT and examines the descriptor. Is the Present bit set to 1? If it were not, the segment would be absent and the CPU would trap out to OS/2.

The *DS* register is being loaded, so the CPU checks that the descriptor specifies a data segment or a readable code segment. If it were execute-only code or one of several special system descriptor types, the CPU would trap out to OS/2.

What is the privilege level of the segment? The CPU checks that it is at least as humble as the current privilege level. If it were prouder than the running process, the CPU would trap out.

When all checks have been passed, the CPU sets the Accessed bit in the descriptor for the segment. Then it copies the Read-Write bit, the size, and the address into the CPU hardware. The descriptor selected by *DS* (or any other segment register) is *cached* in this way, so that the tedious lookup process need only be done once, when the register is loaded.

This process takes 17 CPU clock cycles, or about 2 microseconds in typical hardware. When it is finished the CPU knows the size, 24-bit address, and read-write state of the segment. Whenever an instruction refers to a byte or word at some offset in that segment, the CPU can fetch or store it quickly.

Reference: DGROUP 3.2.4.5

2.4.5.2 Loading a Code Selector

Far jump, call, and return instructions all load a selector into the *CS* register. The CPU goes through a similar process of testing and validation for this selector. It is tested against the size of the descriptor table to which it applies. Then the table is indexed and the descriptor examined to make sure that the segment is present, is a code segment, and has an appropriate privilege level. If all tests are passed, the CPU caches the size and address of the code segment so that it can quickly fetch the next instruction from the offset in *IP*.

2.4.6 Shared Segments

Any segment described in the GDT is automatically common to every program. As mentioned, OS/2 uses the GDT only for segments that should be part of the address space of every program without exception, that is, the call gates into the OS/2 kernel and the data structures it uses. (The data structures have a high privilege level so that they remain inaccessible to application code, even though they are addressable.)

There are lots of reasons for sharing other kinds of segments between programs. Segments of application code and application data, as well as some segments of system code, are shared by way of the Local Descriptor Tables of the sharing processes. When a segment is to be shared, a descriptor for it is entered into the LDT of every process that will have access, but not into the LDT of a process that doesn't need it.

2.4.6.1 Same Selector or Different?

This sounds simple, but becomes complicated in practice. The main question is, will the selector for a shared segment be the same in every process that uses it?

The selector for a segment is actually the offset of its descriptor in the LDT. If the selector is to be identical for every process, then the descriptor must be entered at the identical offset in the LDT of every process. If the selector may differ from one process to another, then the descriptor may be entered at any free position in an LDT.

2.4.6.2 Shared Dynamic Allocations

In the topic on dynamic storage allocation, three methods of sharing dynamic storage are discussed. One of the principal differences between the methods is whether the selector for the shared segment is the same in every process or may be different. It makes a difference in programming. When the selector will be the same, the creating process will be able to tell its clients what selector to use. When the selector differs, the clients will have to find out for themselves what the selector is in their address space.

Reference: allocating shared segments 7.4

2.4.6.3 Shared Code Segments

The sharing of code segments is central to OS/2 operations. All segments of code are read-only, and so can be shared among as many processes as need them. OS/2 tries never to load more than one copy of a code segment, no matter how many different processes are using it. Once the segment is loaded, any additional process that is linked to it simply gets a duplicate descriptor in its LDT.

In order to know what code segments it has loaded, OS/2 keeps a table of loaded segments ordered by module name and segment number within the module. In this context, module means a linked program image; the output of the Linker. A module may be a program file (.EXE file extension) or a dynamic link library (.DLL file extension).

A code segment must have the identical selector in every LDT in which it appears. The reason is that the code segment may contain far pointers to itself. It might contain, for example, a far-call instruction from one procedure in the segment to another. The selector of the destination segment is embedded as an immediate operand in a far call instruction. If the code segment appeared in more than one LDT, but with a different selector in each, then in at least one address space it couldn't execute correctly because its embedded references to itself would be wrong.

When a module is first loaded, OS/2 notes its name and assigns LDT selectors to each segment in it. Those selector numbers are reserved to those segments as long as they are loaded. The first and any subsequent processes that link to that code will have descriptors for those segments in their LDTs. The corresponding positions of the LDT will remain blank in any process that is not linked to the module.

Reference: table of loaded modules 4.1.2.2

2.4.6.4 Shared Data Segments

The segments of data that are linked with a module may or may not be shared. This is established at the time of linking, either by default or by use of a link definitions file.

When a data segment is shared, it is handled much like a code segment. A single copy is loaded from disk just once. Then, a descriptor for that segment, naming it as writable data, goes into the LDT of every process that links to the module.

Such a data segment can be modified from different processes asynchronously. Methods of using and controlling common data are discussed in the topic on asynchronous processing.

When a linked data segment is not shared, a fresh instance of it is created for every process that links to the module. Into the LDT of each process goes a descriptor for its own, unique copy of the data segment.

Whether a data segment is shared or not, it must always use the same selector number in every LDT. The selector number will be embedded as an immediate operand in instructions in the code segments of the module, and they are common to all using processes. The data selector has to be the same in all address spaces or else the code would fail in some processes.

References: table of loaded modules 4.1.2.2; Link options for shared data segments 3.4.4.1

2.5 Privilege Levels

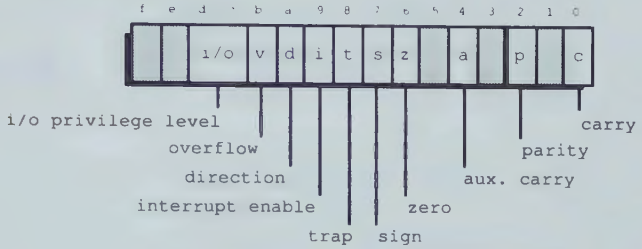
The *privilege level* of a program determines what segments it may access and what instructions it may execute. There are four possible levels:

- zero is the most powerful level, and is required to execute instructions such as the one that loads *LDTR*
- level one is not used in OS/2
- level two is the level required to execute I/O instructions
- level three is the level of most application code

The I/O privilege level is set by two bits in the machine flag register, whose contents are shown in Figure 2.5. Under OS/2 it is always set to two.

The *Current* Privilege Level (CPL) is set by the two least significant bits of the CS register. This is the value that determines what can and can't be done, the value that must be at least as proud as any data that is accessed, and the value that determines what code segments may be called. The initial CPL is established for the application when OS/2 prepares its registers.

Figure 2.5
The Machine Flag-Word



2.5.1 Privilege for Data Access

The rules for access to data have already been mentioned. The privilege level encoded in the descriptor of a segment must be at least as humble (at least as large, numerically) as the CPL. If it is prouder (numerically less), the program may not access that segment as data; if it tries, a hardware trap will end the program.

Privileged code, that is, code running with a proud CPL, may access data with a humble level. The OS/2 kernel may access segments owned by an application. Application code may not access segments owned by the operating system.

2.5.2 Privilege for Call and Return

The rules for calling are more complicated. They are designed to allow applications to call on the operating system (but only in a controlled, restricted manner) and for the operating system to return to the application.

The basic rule is that a far call or jump may only enter a code segment whose descriptor privilege level is exactly equal to the CPL. Humble code may not branch or call into proud code. If it could, the application might subvert the operating system by branching into it to execute some sensitive operation. Proud code may not call into humble code because a call implies a return, and the humble code might subvert the operating system by altering the return address.

It is possible to mark a segment of code as conforming to its caller's privilege. Such a segment may be called when the CPL is humbler than its own described

privilege. The CPU permits the call to take place, but it does not change the current privilege level. If the entered segment can execute correctly at the humbler privilege, fine.

These rules don't affect the usual operations of a program. Normally, all of a program's segments have the same privilege level, so calls and jumps may pass between segments freely.

Reference: link option for conforming segments 3.3.4.5.4

2.5.2.1 The Call Gate Mechanism

There has to be some way of calling from humble code to proud code in a controlled way; otherwise there'd be no way of calling on the operating system for services. The method is a *call gate*. This is a special form of descriptor which amounts to an indirect address.

An ordinary far call instruction contains a new selector for *CS* and a new offset for IP; together these form a far pointer to the first instruction of the called routine. But the CPU may discover, when it looks at the descriptor for the new code segment, that it is in fact a call gate. Normally, the descriptor for a code segment contains the address and length of the segment. A call gate contains instead another selector and another offset.

On a call through a call gate the hardware disregards the selector and offset that were in the call instruction. The selector and offset found in the call gate become the actual far pointer to the called code. Since these are loaded from the descriptor, there is no chance for the program to tamper with them. There's no chance, for instance, for a mischievous (or broken) program to call into the middle of a privileged routine. The call will go to the offset established by the system when the call gate was built, and to no other offset.

Because of its security, the call gate is allowed to select a code segment with higher privilege than itself. The gate descriptor itself has a humble privilege level (so that humble code may access it) but the destination it leads to may be privileged. The old, humble privilege level is pushed onto the stack along with *CS*, as part of the return address. The new privilege level is adopted so long as control remains in the called segment. During the return the old CPL is resumed.

References: link option for I/O privilege 3.3.4.5.4; link option causing a call gate 3.3.4.7.1

2.5.3 The Task State Mechanism

The privilege mechanism is based on a special block of storage called a *task state segment*. The main use of this segment is to store the registers of the running process when an interrupt occurs. The hardware stores registers into the task state segment on an interrupt; it also reloads them when the operating system restarts the process.

Task state segments are set up by the operating system. Their initial contents determine the register values with which a process will begin execution (including its privilege level, as part of its initial *CS*).

While a task state segment has single values for for most registers, it actually specifies different values of *SS* and *SP* for each of the four possible privilege levels. And here's the trick: when a call through a call gate causes a change in CPL, the hardware automatically picks up a different stack space from the task state segment! This isolates the stack used by privileged code from the stack used by humble code. The old stack is resumed as part of the change back to humble privilege in the return.

Since privileged code uses a different stack, there has to be a means of getting the privileged routine's stack parameters from the humble stack to the privileged stack. And there is: in the call gate descriptor itself is a number that specifies how many words of parameters must be copied during a call. That many words, plus the return address, are automatically copied from one stack to the other during the execution of the call instruction.

2.5.4 I/O Privilege

When an application program is linked, it may designate any code segment as needing I/O privilege. The Linker marks such a segment so that when OS/2 enters it to the LDT it sets the descriptor privilege level to 2. The procedures in the segment may execute I/O instructions and may set and clear the interrupt-enable flag. This provides a way for application code to manage hardware devices such as the video graphics adapter.

However, the rest of the program will be running with the usual CPL of 3. Direct calls from level 3 to level 2 are not permitted by the hardware. There has to be some way of getting into these privileged routines. It is possible to have them marked conforming when they are linked, but that only allows them to be entered with the CPL of 3, under which I/O instructions cannot be executed.

The only solution is that OS/2 must set up a call gate to each entry point in the privileged segment. A call gate can only lead to a single procedure, so one has to be built for every public entry point in the segment. The means of creating these call gates is discussed in the topic on linking.

References: link option for I/O privilege 3.3.4.5.4; link option causing a call gate 3.3.4.7.1

3

Linking

The topics of *linking* and *loading* are intimately connected, and each has to be understood in terms of the other.

One of the central functions of the OS/2 kernel is to load a program from disk into storage and prepare it for execution. Let's name the part of the system that does it the Loader.

The Loader requires a program to be in a special format. The *LINK* utility prepares files in that format, using the output of a compiler or an assembler as its raw material. Let's call this complicated, useful program the Linker.

This long chapter explores the Linker. The next covers the Loader.

3.1 Linking and Loading

Stand back and think about the whole process of Linking and Loading from end to end. A program begins life as source modules in some language or other—at least one module but possibly dozens or hundreds of them, because OS/2 can support very large, industrial-class software projects. At this stage, the connections between different program units are purely symbolic, in the form of language commands and keywords.

The source files are compiled or assembled into as many *object* files, that is, files in the binary format that the Linker supports. There are usually libraries containing runtime support modules, also in object format. At this stage the connections between units are still symbolic, in the form of character-string names of external and public labels embedded in the object files.

The Linker combines object files into fewer, larger units that correspond exactly to the segments of program code and data to be loaded into storage for execution. The Linker's output (in a file with the extension .EXE or .DLL) consists of *binary images* of the program segments—near-perfect copies of how the program must appear in storage if it is to execute. All that's lacking is the final assignment of selector numbers to segments; that has to wait until load time. At this stage most connections between units have been reduced to numbers, the ordinal numbers of the different segments. Some connections, however—the *dynamic links*—have been left in the form of character strings.

At execution time the program has to be an OS/2 process, which is a collection of code and data segments in storage described by segment descriptors in a Local Descriptor Table (LDT). Now all the connections between units must be absolute binary selector numbers and offsets that are embedded in the code.

This is really a single process, not a series of different ones. It's a process of reducing the program from many small units to a few monolithic ones, and of stripping away its readable, symbolic content and replacing it with binary data.

The process is divided into phases called Compiling, Linking, and Loading only for efficiency. The earlier steps are the hardest, and are done the least often. The later steps do the least work and happen most often. The boundary between phases could be located at different points, with more or less work done at Load time. In fact, the boundary between the Linking and Loading is located differently in OS/2 than in MS-DOS; some steps that are Linking in MS-DOS are Loading under OS/2.

Here we launch into a lengthy discussion of the LINK program. The Linker is a highly complex program with many options. It is absolutely central to a programmer's work, no matter what language is used. Its abilities and methods govern the operations of the compilers and assemblers that feed it, so when the Linker is understood the restrictions of other programs may seem less arbitrary.

3.2 An Overview of Linking

The result of compiling or assembling one file of program source is a file of *object text*, which is binary data that represents the same program statements translated into machine language. It is just conceivable that a solitary object file could represent a complete, useful program, but this is almost never the case.

Almost every object file needs to be *linked* to other object files to make a complete program.

Some of the other files may represent other modules of the program—large applications are often split into dozens of separate modules. Some come from the *runtime library* supplied as part of a programming language, and contain the standard subroutines defined as part of that language. And some of a module's *external references* may be to subroutines that are a part of the OS/2 operating system. (A compiled module might not refer to system routines, but the runtime support routines it calls surely will.)

The basic job of the Linker is to merge all the object modules that compose a program, resolve the linkages between them, and prepare a finished *program image* in the form of an *executable file* that the Loader can load and start.

In OS/2, it is possible to leave some of a program's external linkages unresolved until the moment it is loaded. This is called *load-time dynamic linking*. Packages of subroutines are prepared in advance; these are called *dynamic link libraries*. Only one copy of a dynamic link package is needed on disk and in storage no matter how many programs link to it. The most important dynamic link packages are the ones that make up the OS/2 system itself. All of a program's references to system entry points are resolved to dynamic link entry points by the loader.

A dynamic link package itself is kept on disk in the form of an executable file. The file extension of a dynamic link library is .DLL instead of .EXE, but the format of the file is almost identical to that of a program's executable file. It differs primarily in that a dynamic link file contains a table of entry points that shows the system where a *client program* may link into the package.

The Linker is used to prepare a dynamic link package, and the process of linking one is almost identical to linking a normal program. The object modules that go into a dynamic link package are created by compiling or assembling source files and have the same format as other object files. During linking, a *definition file* tells the Linker how to build the table of entry points for the package.

3.2.1 The Object Module

In order to understand the Linker we must first understand its input data, the contents of an object module.

3.2.1.1 Essential Segments

Underneath all the confusion of linking there hides one simple notion: the concept of an *essential segment*. You won't find the term in any OS/2 manual because the concept is so simple it is taken for granted. It will help to make it explicit.

An essential segment is a sequence of bytes whose contents are defined by the program and which at execution time must be:

- adjacent
- in their defined sequence
- addressable

or else the program will not execute correctly. If you can examine a program and say of any two bytes that the program will fail unless they are located at a specified distance from each other, then those two bytes (and whatever unspecified bytes lie between them) are part of one essential segment. Conversely, if you can say of two bytes that it doesn't matter where they lie in storage relative to each other, then those bytes are in different essential segments.

Regardless of its source language, a program is reduced by compilation or assembly to a set of essential segments and some supplementary information about how these are related.

Consider a subroutine: it is translated into a sequence of machine instructions. At execution time the bytes that encode those instructions must be adjacent (no gaps may be inserted during linking or loading), still in their sequence (if they are disordered the program can't run), and addressable (if the program can't address them, it can't call the subroutine).

Consider a word of data initialized to an integer value. The program will contain instructions to load that word into a register; if the two bytes of the word aren't adjacent and in their defined sequence the program will fail. The same is true of the bytes in larger data structures.

Reference: register 2.2

3.2.1.2 Defined Segments

Essential segments are generally quite small: a single subroutine, a single variable. It would be possible in principle to make each one into a hardware segment when the program was loaded. The only reason this isn't done is that it would be inefficient. A large program would have a huge number of segments, perhaps more than a Descriptor Table could hold. Every call would have to be a far call, and a segment register would have to be loaded before every reference to a variable in storage.

For efficiency, programs usually lump their essential segments into a smaller number of *defined segments*. These are the segments that are defined with segment directives in assembler language. A typical assembly language module will specify one data segment that contains all the variables defined in the module, and one code segment that contains all procedures defined in the module. The typical result of compiling one module of C or Pascal code will also be a single segment of data and one of code.

Defined segments usually contain a number of essential segments; that is, they contain a number of procedures or variables that don't really need to be in any special relationship to each other. However as soon as a defined segment is created, it *becomes* an essential segment. It is made essential because the machine instructions in the object file will contain the offsets of objects within their segments. An instruction to load a register from a variable will assemble into an opcode byte and the byte offset of the variable in its defined segment. A near call instruction from one procedure to another will contain the offset from the call instruction to the called point. All the separate objects in the defined segments are locked into their relative offsets, not because of logical relations among them, but because all the instructions that refer to them would fail if they moved.

The compiled program places three requirements on the contents of a defined segment: two requirements it placed on an essential segment and one new one. The bytes of a defined segment must be:

- adjacent
- in their defined sequence
- addressable from a single segment register

or the program will fail. These three expectations are built into the compiled machine instructions, and the Linker and Loader are required to respect them.

References: descriptor table 2.4.4; segment register and offsets 2.2.4

3.2.1.3 Bindings Between Object Units

The units of an object program are woven together by a network of references from each unit to others. These connecting references are in general called *bindings*. When the program is in source form, all of its bindings are *symbolic*, that is, expressed in text symbols and not in machine addresses. Before the program can be executed, all of its bindings must have been *resolved* to binary machine addresses. Some of the symbolic information may be kept (for example, as data to help CodeView display the source file) but the effective bindings must all be binary pointers that the CPU can handle.

The object module contains two types of information. A large part of it is simply the static, binary contents that the defined segments are to have at execution time. This is the binary image of the defined segments, a picture or a template showing how they must look in storage when executing. It includes

- The binary text of machine instructions and their immediate operand values.
- The binary text of program constants and literals.
- The binary text of initialized variables.
- The sizes of blocks of uninitialized data space.

The rest of the information in an object file consists of records that define the bindings between defined segments and between that object module and others.

3.2.1.3.1 Bindings Between Segments of a Module

Bindings between the defined segments of a module are recorded so that the Linker can combine segments. Separate records point out each spot in the binary image where there ought to be the selector number of one of the module's segments. For example, if the binary image contains the sequence

```
mov ax, seg dgroup
mov ds, ax
```

the first instruction will have as its immediate operand a word that will contain only a zero in the binary image, since the selector for the *DGROUP* can't be known at compile time. The same is true of a selector in a data word, as in

```
dw  seg dgroup
```

within a data segment. In either case a separate object record will say, "at such an offset in the code segment please write the selector for my defined segment *DGROUP*."

Other records point out every spot in the binary image where there is now an offset into one of the segments. There are offsets to data in many instructions; for each one there will be a separate object record that says, "here is an offset to segment such-and-so." These records will enable the Linker to combine the defined segments of different modules.

References: combining segments 3.2.3; *DGROUP* 3.2.4.5

3.2.1.3.2 Bindings Between Modules

The bindings between object files are symbolic. The names of objects that a module *exports*, that is, makes available to others, are called *public* names. The names that a module *imports*, that is, the names of objects that it requires others to define for it, are called *external* names.

3.2.1.3.2.1 Defining Exported Objects

A *public* object is an object that one module defines and makes available for use in other modules. The object can be an item of data (either constant or variable) or the code of a procedure or function. All that is preserved in the object module is the object's name and its offset within its defined segment. All other typing information (such as the size of the object, its data type or structure, whether a procedure is meant for near or far calls) is left behind during compilation.

Public names are defined in assembly language with public statements. They are defined implicitly in higher languages. In Microsoft Pascal, all names declared at the outermost level of a module or implementation file are public. In addition, procedures and variables in any Pascal source file may be given a public attribute that causes them to be noted as public in the object file.

In C, all functions and all global variables are public by default. The *static* keyword can be used to keep names private to their source module; otherwise they will be public in the object file.

3.2.1.3.2.2 Defining Imported Objects

An *external* object is one that the module expects will be defined by some other module with which it will be linked. What appears in the object file is the external name plus pointers to all the places in the binary image where the selector or offset of the named object ought to appear. Whatever expectations the programmer may have about the imported name (its data type or size or whether it's a near or far procedure) are left behind in the source file.

In assembly language an external label is defined with an *extrn* statement. This statement specifies whether the name represents data or a procedure. If it's data, the statement specifies its type; if a procedure, it specifies whether it's near or far so the assembler can generate the appropriate type of call or jump instruction.

In Pascal, external procedures and functions are declared with the *external* directive. In Microsoft Pascal, variables too may be declared with an *extern* attribute.

In C, undeclared functions are assumed to be external and to use the parameters that are programmed into the calls to them. However, the Microsoft C compiler supports the proposed ANSI standard; it permits external procedures to be declared with the *extern* keyword and with symbolic parameter lists. If this is done, the compiler can check the call parameters for number and type.

In all these cases, information about data types and nearness or farness is used by the language processor to generate code, but is not passed into the object file. As a result, it can't be checked by the Linker. If the importing module believes that external name *FRED* is a far function returning an address, and the exporting module actually defines *FRED* as an integer variable, the error will not be caught until execution time.

3.2.1.3.2.3 Defining Common Segments

Assembler and FORTRAN support yet another kind of intermodule binding: the sharing of common data segments. There may be any number of these. Each has a name. Each will be created by the Linker when the program is linked. In a sense, common segments are external to all modules.

The object module defines a common segment the same way it defines a segment of data: with a name and with a binary image. What distinguishes a common segment from a normal one is that it has a combine-type of *COMMON*. (Segment

combine-types in general are discussed later, but the handling and use of common segments deserves this separate discussion.)

The image of a common segment is normally empty; only its expected length appears. However, one module may include an image of initial data for the segment, or different modules may define initial values for different parts of it.

The Linker will probably receive a definition of a common segment, from each object module that uses it. It will create a single segment; whose length will be the longest one specified by any definition. Its contents will be formed by laying all binary images into the one segment. If more than one module contains initial data for the same part of the segment, there's no telling which image will prevail.

FORTRAN compilers use common segments to implement the *COMMON* declaration. Assembler programmers are free to define them using the *COMMON* segment combine-type (more about this later). Some C compilers use common segments to implement global variables, but the Microsoft C compiler uses a different mechanism.

3.2.1.3.2.4 Defining Communal Names

The Linker gives special support to the needs of C compilers. In C, a variable that has external scope (is global or declared *extern*), but which lacks an initializer, is generally expected to be defined outside the module. If there is no defining instance of the variable among all the modules that name it, it has still to be defined as a segment or part of one.

Different C compilers handle the binding of such variables in different ways. One method is to use common data segments. The compiler defines a common segment for each external global variable, including the image of initializing data if any is specified. The Linker handles these as it would any common segments, merging all the definitions of one variable into one common data segment.

Other compilers use the mechanism of exported and imported names. These compilers must place strict rules on the use of the keyword *extern*. The programmer has to declare a variable as *extern* in all but one source module of the program. When it sees *extern*, the compiler creates a reference to an external name. Exactly one module must declare the variable without *extern*, and only that module may declare an initializer for it. The compiler, on finding a global variable declared without *extern*, defines the variable in a data segment and creates a public name for it that other modules may link to.

The Microsoft C compiler uses still a third method. When it compiles a global variable that lacks an initializer, it generates a *communal name*. This special object record defines the name and the expected size of the variable, but does not locate it in any segment and does not specify a binary image for its contents.

When Microsoft C compiles a global variable with an initializer, it locates the variable in the data segment and generates a public name for it.

The Linker handles communal name records as tentative external references. When a name is defined both in communal name records and as a public name, the Linker simply treats the communal references as if they were external references to the public name. If no source module contains an initializer for a global name, the Linker will receive only communal name records for that name. The Linker disposes of such names by creating a single data segment named *c_common* and defining uninitialized space in it for any communal name not defined elsewhere. In effect, the Linker tidies up the loose ends left by the C programmer.

Communal variables may be defined in assembler language using the *COMM* statement.

3.2.1.3.2.5 Defining Dynamic Link Names

A dynamic link package is not bound to its client program until the program is loaded. Nevertheless there has to be something to tell the Linker that a dynamic link package *will* exist and will contain certain public names.

An entry point to a dynamic link package is represented by a special object record. A utility called Implib produces the object file to represent a dynamic link package. It contains nothing but the names of the entry points to the package, and the name of the package. The DOSCALLS library distributed with OS/2 is one of these files.

When the Linker reads one of these artificial object files it simply notes the names as exported names that will be available to the Loader.

3.2.1.4 The Program Entry Point

The object file may contain a record that specifies the entry point at which the program should begin execution. If more than one module tries to specify the program entry point, the Linker will complain.

In assembly language the entry point is defined by naming the entry-point label on the *end* statement of the module. Compilers generally do not define a program entry point, not even for a *main()* C function or the main begin-block of a Pascal program. They leave it to some module of runtime support that is linked by all programs (it's *5crt0* for Microsoft C and Pascal) to define the program entry point. That's how the runtime procedure that sets up the execution environment gets control to set up the *argc* and *argv* parameters in C, for example, or to open the default text files in Pascal. When it finishes, it calls an external procedure named *main*, and that's how the compiled code receives control.

3.2.1.5 Default Libraries

One special record of the object file gives the filename of one or more object libraries that are to be searched. Compilers use this record to specify the library names of runtime support code. It's purely a convenience for the user, since the libraries could just as well be named to the Linker with command parameters.

3.2.1.6 Object File Format

All of this information is encoded in a peculiar file format that was initially specified by the Intel Corporation. The format used in OS/2 has been extended and revised from the Intel form by Microsoft Corporation.

The Microsoft Object File format is an exceptionally complicated and difficult format. It was apparently designed with the overriding objective of conserving disk space without regard to cost in processing time or program complexity. Worse, it is incompletely documented; certain record types used by Microsoft compilers are considered proprietary. Fortunately, the only people who have to understand the format are those who write or maintain compilers or assemblers. It is not documented here.

3.2.2 How Bindings are Resolved

The Linker reads one or more object modules and notes the information in them: the names and sizes and binary images of all the segments they define, and whether those are code, data or common; all the public names they define; all the external names they reference; and all the communal names they declare. We may suppose that it gives each defined segment an identifying number at this point. The boundaries between object modules don't matter after this; the Linker simply works on a big collection of defined segments and binding records.

The Linker resolves bindings by matching name-strings. An external name is matched to the public name with the identical spelling. The public name will be defined in regard to some defined segment and offset, and this information can be propogated to all the places where it is needed; that is, the external references can be pointed to the defined segment and offset of the public name.

Sometimes an external name matches a public name from a dynamic link package. Then the Linker can't know the segment and offset. It leaves notes in the executable file telling the Loader what names are needed, and which packages define them.

The first time the Linker meets the name of a common segment, it treats it as a data segment. When another definition with an equal name arrives, it is merged with the first one. The segment is given the longer of the two lengths, and the later binary image is overlaid on it, perhaps replacing all or part of a previous one.

The binding of communal names has been described. In effect they are external references, but with the additional service that if no object module defines a public name to satisfy them, the Linker will do so.

> **References:** fixing imported references at load time 4.1.2.3.2; common segments 3.2.1.3.2.3; communal names 3.2.1.3.2.4; letter case when matching names 3.3.3.3

3.2.3 How Segments are Combined

When all the modules of a large program are linked there might be hundreds of defined segments. In principle these could be allowed to stand, but for efficiency's sake the Linker will combine them wherever possible.

The rules for combining segments are complicated, although a bit simpler in OS/2 than in MS-DOS. There are general rules that can be exploited by the as-

sembly language programmer. Then there are the conventions that the Microsoft language products use to implement their memory models. These conventions are important because they must be followed by assembly language modules that will be linked with a compiled program, and because they explain some limitations of language implementations.

3.2.3.1 The Attributes of Segments

Every defined segment has four attributes that tell the Linker how to handle the segment and how to combine it with others. These are the segment's alignment, its name and class, and its combine-type. In assembly language the programmer has to specify each of these. In high level languages the compilers generate these four items for each defined segment.

3.2.3.1.1 The Alignment Type

The alignment type of a segment tells the Linker and Loader whether the program expects the first byte of the segment to be aligned at any particular storage boundary at execution time. The possible alignments are:

- *byte*, any boundary will do
- *word*, starting address to be an even number
- *dword*, starting address to be a multiple of 4
- *para*, starting address to be a multiple of 16
- *page*, starting address to be a multiple of 256

Code segments generally specify *byte* alignment, since machine instructions don't benefit from any special alignment.

Data segments generally request either *word* or *dword* alignment. The CPU can always fetch and store *word* operands faster on even-numbered boundaries. Systems that use the 80386 generally have 32-bit-wide RAM; in these machines the CPU can fetch word and four-byte operands faster from addresses that are a multiple of 4. By requesting *word* or *dword* alignment, and by maintaining that alignment within the segment, the program can be sure that all its operands will fall at advantageous addresses in storage.

The *para* alignment type is often used in MS-DOS. When a segment is aligned on a multiple of 16, a program in Real mode can be sure that the offset of the first item in the segment is zero. (With *word* alignment, the first offset might be

as large as 14.) In Real mode it can be useful to know this. In Protect mode the first offset in a segment is always zero and arithmetic on selectors is impossible, so *para* alignment is of no benefit. *Page* alignment is also rarely used.

The alignment type of a defined segment will be maintained when segments are combined, so that, for instance, a segment with *dword* alignment will be set on at least a doubleword offset relative to the start of the physical segment in which it ends up. Physical segments are loaded on page boundaries.

3.2.3.1.2 Segment Name and Class

The name of a segment is carried in the object module as a character string. The class of a segment is a second character string that qualifies the name, in much the same way that a file extension qualifies the name of a file. The Linker will try to combine any segments that have identical names and classes.

It is conventional for the class to suggest the use of a segment and the name to specify a unique segment within the class. Generally the class may be any string the programmer or compiler chooses. The class *CODE*, however, and any class that ends in the letters *CODE*, signals a code segment to the Linker, the Loader, and to CodeView. Any segment with that class will be treated as a code segment, and segments with other classes will not.

3.2.3.1.3 The Combine-Type

The combine-type of a segment specifies whether the Linker may try to combine it with other segments or not. If it is omitted, the segment is said to be private and may not be combined with any other. If it is *COMMON*, the segment is a common segment. The processing of common segments is discussed above.

When the combine-type is specified as *PUBLIC*, the segment may be combined with any other segment that has the same name and class.

When the combine-type is *STACK*, the segment may be combined with any other of the same name and class, and in addition the (combined) segment is the one whose selector will be placed in *SS* when the program is started.

At least one segment in every program must have the combine-type *STACK*. Only Tiny-model programs can do without a stack segment, and the Tiny model

isn't supported by OS/2. (This rule does not apply to dynamic link packages, since they are not complete programs and will probably use their client's stack.)

3.2.3.2 Combining Code Segments

When the Linker finds two segments that have:

- identical names
- the class CODE
- the PUBLIC combine-type
- an aggregate size not exceeding 64K bytes

it combines them. It's not hard to combine segments; they may simply be slapped together end to end and called one. Code segments contain near jumps and calls that contain relative offsets; that is, operands that are the distance from instruction to destination. These need no adjustment.

Of course, if there are far pointers to the second of the combined segments, they must be adjusted. There might be far pointers to the segment within the segment itself, or in other defined segments. Their selector portions must be noted as referring to the combined segment. Their offset portions, and any other near pointers relative to the second combined segment, must be increased by the length of the first combined segment.

After these adjustments have been made, it's as if there had always been the one large defined segment. The three requirements that the program makes have not been violated: the bytes it defined are still together, in sequence, and addressable from one segment register.

After code segments are combined there's an opportunity for a small speed optimization. If there was a far call from one of the combined segments to the other, it no longer needs to be a far call. It can be left as a far call; the loading of the *CS* register won't do any harm. But there is an equivalent instruction sequence that is possible in combined segments:

```
nop
push cs
call (relative offset)
```

The one-byte *nop* (no-operation) instruction brings the sequence to the same length as a far call. The sequence takes just half the time of a far call (13 clock cycles versus 26).

If requested to do so with the */FArcalltranslation* option, the Linker will make this change wherever it finds a far pointer in code preceded by the opcode value of a far call. The service is optional because it is just possible that a byte sequence that looks like a far call might be some kind of static data that includes a far pointer.

> **References:** near and far calls 2.2.4.2.1; Code Segment register 2.2.4.2; for call option 3.3.3.8

3.2.3.3 Combining Data Segments

When the Linker finds two segments that have:

- identical names
- identical classes other than *CODE*
- equal combine-types of *PUBLIC* or *STACK*
- an aggregate size not exceeding 64K bytes

it combines them. The combined segment will be given the more stringent alignment type of the two, and the Linker may have to insert undefined bytes between the segments in order to maintain the alignment of the second one.

As with code segments, the Linker has to adjust all far and near pointers that refer to the second of the two segments. These may appear in any segment. The selector portions of far pointers must be noted as referring to the combined segment. Their offset portions, and any other near pointers, must be adjusted upward to account for the new offset of the first byte of the second segment. There will also be offsets to the second segment embedded within instructions in code segments, and these have to be adjusted.

When all the adjustments have been made, it's as if there had always been the one defined segment. The program's requirements have not been violated: the bytes it defined are still adjacent, in sequence, and addressable.

The Linker continues to combine code or data segments until it has made all the combinations it can and has reduced the number of defined segments to the smallest number it can.

3.2.3.4 Forming Groups

A program may specify that it wants certain defined segments to be placed in a *group*. A group is a collection of defined segments that must all be addressable using a single selector, but which need not be in any particular order. A group is another efficiency dodge; it's a way of cutting down on the number of different selectors the program has to know and on the number of times it has to load a segment register.

In assembly language, a group is specified with a *group* statement. In it, the programmer specifies a name for the group and lists the names of the segments that are to be part of the group. The Microsoft compilers specify a single group named *DGROUP* in which they keep the most important program data.

The group definition is carried as a record in the object module. The grouped segments don't have to have any particular names, classes, or combine-types. In Protect mode it isn't feasible to group code and noncode segments, since the hardware won't allow code and data to be used from the same selector.

The object file documents every offset in the binary image and the segment to which it is relative. It is also possible to have an offset that is relative to a segment within its group. When the object file is created, all that can be known is the offset within the segment; the position of the segment within its group can't be known until the group is formed.

After it has finished combining segments, the Linker forms groups. It puts the member segments together end to end, inserting bytes where necessary to maintain alignment. Since a group must be addressable from a single segment register, it will in fact be a single segment when the program is loaded, and of course it may not total more than 64K bytes. Now that the Linker knows the offset of each segment within the group, it can go back and adjust the offsets that were relative to the group.

Reference: DGROUP 3.2.4.5

3.2.3.5 Packing Physical Segments

The programmer (or compiler) directs the combining of segments by name and group, and can take advantage of it. Even after this has been done, there are opportunities for packing defined segments and groups into physical segments.

Loading goes faster, and the OS/2 kernel's tables are smaller and simpler, when a program consists of fewer, larger segments.

The Linker can't merge data segments without the program's knowledge, since it is possible that the program will use the *DosReallocSeg* call to enlarge one of the segments at run time. That wouldn't produce the expected results if the segment the program tried to enlarge wasn't a unique physical segment.

But that danger doesn't apply to code segments. (*DosReallocSeg* might or might not extend one, but there's no point in trying because, since code segments are read-only, the extended space could never be used.) So long as the Linker can properly adjust all far and near pointer values, there's no reason it shouldn't pack separate code segments into single physical segments regardless of their segment names and classes.

The processing of packed code segments is the same as already described for combined code segments. It is not done to code segments that are named in a *SEGMENT* list in a definition file (discussed later), and it can be prevented entirely with the */NOPack* option or by naming the segment in the definition file.

> **References:** *DosReallocSeg* 7.3.6; no-packing option 3.3.3.8; segment statement in definition file 3.4.5

3.2.3.6 Segment Order

The order in which the Linker wrote segments to the .EXE file was quite important under MS-DOS, because in Real mode a program could calculate segment addresses by arithmetic from other segment addresses. Questions like "which segment is the lowest" or "the highest" or "where is the end of the program" can be given meaningful answers in Real mode.

In Protect mode, these questions have no meaning. There is only one valid selector for any segment, and it has no numerical relationship to any other selector. There isn't any order of higher or lower or first or last among the segments of a Protect mode program.

The Linker can be directed to use a particular order of segments, either by a command option or by a record embedded in an object file. This DOS segment order puts code segments first, far-data segments next, and the segments of the *DGROUP* last in the .EXE file. Most compilers still place the DOS Order record in their object files, and it is possible that CodeView may expect .EXE files to be in DOS Order. It does no harm to specify it.

DOS Segment order also governs the order of segments within *DGROUP*. This order is discussed in the next subtopic.

References: read vs. protect mode addressing 2.4; DGROUP 3.2.4.5

3.2.4 Linker Support for Compilers

The Microsoft languages receive special support from the Linker and the Loader. This support is based on using segment and group names, classes, and combine-types in certain conventional ways. The conventions are followed by most compilers, and by the OS/2 assembler when simplified segment directives are used.

The conventional values are listed in Table 3.1, in the order that the Linker will put them when using DOS Segment Order.

3.2.4.1 Conventional Code Segments

Programs using the Small or Compact memory models have only one segment of code. Its name will be *_TEXT* and its class will be *CODE*. Programs that use other models will usually have a *_TEXT* code segment as well, since the code of library support modules will usually be defined in that segment.

Separately-compiled source modules for Large-model programs will produce code segments named *module_TEXT*, where *module* is the filename of the source and object files. Since these segment names differ from each other and from *_TEXT* alone, they will not be combined by the Linker (although they may be packed together).

Regardless of name of a code segment, it should always have the class *CODE*, or at least a class name with *CODE* as its last four letters. This will cause the Linker to place the code segment adjacent to the other code segments in the output.

References: memory models 2.3; segment class 3.2.3.1.2

3.2.4.2 The ENDCODE Class

A segment with the class *ENDCODE* may appear last among the code segments. Since its class name ends in *CODE*, it will be ordered with code segments. In Real mode some programs may use its address as marking the end of the code and the start of data. In Protect mode, of course, its address has no arithmetic relationship to that of any other segment.

3.2.4.3 The BEGDATA Segment

There is usually only one definition of a segment with class *BEGDATA*. It will be found in a runtime support module that all compiled programs include. Its name is the null string. Its binary image is usually a copyright statement.

The Linker will ordinarily put this segment first among data segments. In Real mode this puts it just after the last code segment in the numeric sequence of storage addresses, and just before the first normal data segment. In Real mode some library modules may rely on this positioning. Of course, in Protect mode there is no arithmetic relationship between segment addresses; it is pointless to make assumptions about the location of segments in storage.

3.2.4.4 The Far Data Segments

Modules compiled to the Medium, Large, or Huge models will usually have multiple segments of data. When the data is initialized, it will have the class name *FAR_DATA*. When it is not initialized, its class is *FAR_BSS*.

When a source module is compiled to the Medium or Large model, it will generally produce one data segment in each of these two classes. The segments will be named *module_DATA*, where *module* is the same as the filename of the source and object files.

A program compiled to the Huge modal may have several data segments. All but the last will be exactly 64K bytes (10000h) in length. Their classes will be *FAR_DATA* or *FAR_BSS*. Their names will be composed from the module name, a digit, and the suffix *_DATA*.

Table 3.1
Conventional Segments In DOS Segment Order

Segment Name	Segment Class	Combine-Type	Alignment Type	Group Association
_TEXT	CODE	PUBLIC	BYTE	
*module*_TEXT	CODE	PUBLIC	BYTE	
C_ETEXT	END_CODE	*private*	BYTE	
null	BEGDATA	*private*	PARA	
*module*_DATA	FAR_DATA	*private*	PARA	
*module*_BSS	FAR_BSS	*private*	PARA	
_DATA	DATA	PUBLIC	WORD	DGROUP
name	DATA	PUBLIC	WORD	DGROUP
CONST	CONST	PUBLIC	WORD	DGROUP
_BSS	BSS	PUBLIC	WORD	DGROUP
name	BSS	PUBLIC	WORD	DGROUP
c_common	BSS	PUBLIC	WORD	DGROUP
STACK	STACK	PUBLIC	PARA	DGROUP
$$SYMBOLS	DEBSYM	*private*	BYTE	
$$TYPES	DEBTYP	*private*	BYTE	

3.2.4.5 The DGROUP

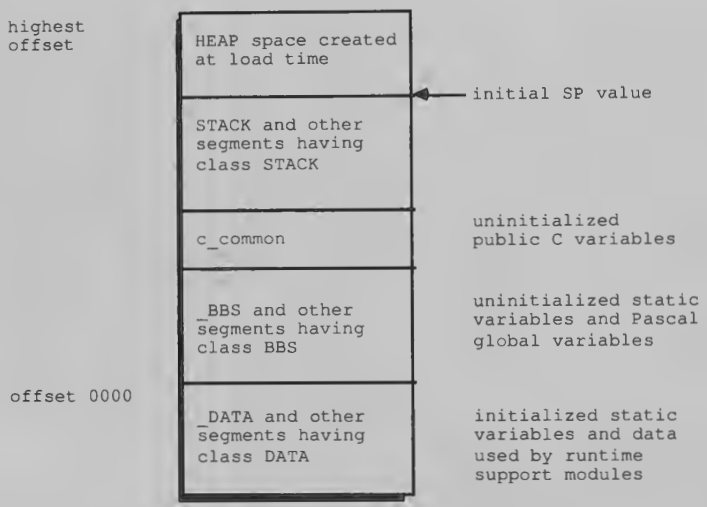
A group named *DGROUP* is always defined by Microsoft languages, and by the OS/2 assembler when simplifed segment directives are used. *DGROUP* contains the miscellaneous constants and global variables defined by the main program and by library support routines. It contains the program stack. And in Small and Medium model programs, it contains the heap as well.

In Table 3.1, the segments associated with *DGROUP* are listed in the order that the Linker writes them. In Figure 3.1 the layout of *DGROUP* is shown as it will be when the program begins execution.

3.2.4.5.1 Data Segments in DGROUP

Compiled modules almost always define a segment with name *_DATA* and class *DATA* to hold initialized static variables. Runtime support modules often define a *_DATA* segment for their own private variables. Some runtime modules define other small segments with class *DATA* and specify that they are to be part of *DGROUP*. The Linker puts all segments with class *DATA* first in *DGROUP*.

Figure 3.1
The Contents of DGROUP



Compiled modules sometimes define a segment named *_BSS* with class *BSS* to hold uninitialized static variables. (Assembly language programs don't usually distinguish between initialized and uninitialized variables.) Runtime support modules sometimes define the same segment for their data, but some runtime modules define segments with other names and class *BSS* for the same purpose. The Linker puts all segments with class *BSS* next in *DGROUP* after those with class *DATA*.

If any communal names were left undefined, the Linker creates a segment to contain them. It names the segment *c_common* (lowercase letters, unlike most conventional segment names) and gives it class *BSS* since it contains uninitialized variables. It ordinarily appears after the last defined *BSS* segment.

It appears that the Linker introduces a public label *_end* into programs that define a *DGROUP*. At least, that label will appear among the public names in the link map file, whether a program declares it or not. Its value is the length of the *DGROUP* through the last *BSS* segment; in other words, its value is the offset of the first byte of stack space. You should not write code that relies on this label; at least, not unless you can find some kind of documentation stating that the Linker will always produce it.

3.2.4.5.2 The Stack Segments

For compiled programs, some runtime support module that all programs link will define a segment with name and class of *STACK* and a default size (typically 2048 bytes). The same module will define the segment as a member of *DGROUP*. Ordinarily no other module will define a stack segment.

The Linker places any segments with class *STACK* last in *DGROUP*. The Linker can be told (with the */STACK* option) to make the aggregate size of these segments larger or smaller.

3.2.4.5.3 The Heap Space

When using the Small or Medium model, Microsoft compilers locate the *heap*, which is the pool of space from which the C function *malloc()* and the Pascal procedure *NEW* draw space, at the end of the *DGROUP*.

It is not necessary for the *DGROUP* segment to contain any heap space when the program begins execution. Any segment may be extended in size using the *DosReallocSeg* call. When called to allocate a dynamic variable, the runtime support for C and Pascal checks to see if there is room. If there is not, the support code tries to extend the *DGROUP* to make a larger heap.

The heap in *DGROUP* is what the Pascal manuals refer to as the near heap. The large-model C function *malloc()* and the Pascal procedures that allocate space in the far heap create new segments of storage using *DosAllocSeg*, and allocate space from within them.

References: *DosReallocSeg* 7.3.6; *DosAllocSeg* 7.3.2

3.2.4.5.4 Use of the DGROUP

Notice that *DGROUP* is ordered so that all initialized data comes first, and that the highest offsets contain only uninitialized space—*BSS* segments, stack space and the heap. The Linker does not write uninitialized space into the *.EXE* file, thus conserving disk space and load time. It does specify the size of the segment; the Loader will make it that size in storage. This makes it easy to implement Link parameters for the size of the stack and the size of the heap. They affect only two numbers in the *.EXE* file, the initial offset for *SP* and the allocated size of *DGROUP* respectively.

In programs compiled to the Small and Medium models, the *DGROUP* contains the only data space in the program. Its selector will be placed in both *DS* and *SS*. As a result, all near pointers to data are relative to the same selector. The program doesn't have to distinguish between a pointer to a global variable (in a *DATA* or *BSS* segment), a pointer to a local variable of a function (on the stack), and a pointer to a dynamic variable in the heap.

When working to the Small or Medium model, compilers may generate code that *assumes* that *DS=SS* and that *DS* has already been loaded with the address of *DGROUP* when the program is entered. Even a public or far procedure will assume that *DS* has been loaded and so will not load it on entry. Moreover, either generated code or runtime support code may need to modify *DS*, but quite likely will not save it, and will quite likely restore it by pushing *SS* and popping *DS*. These assumptions cause no harm in the context of a compiled program. However, they can cause severe problems when compiler-generated code is extracted for use in a different context, for instance as part of a dynamic link library or a subroutine library.

References: link options for stack size 3.3.3.9, 3.4.3; *SP* register 2.2.3; memory models 2.3; segment registers 2.2.4

3.2.5 Debugging Information

Debugging information can be inserted into an object file in several formats. Line-number records relate source file line numbers to the code that was generated from them. These records are much like like the records that define public names: they relate a point in a code segment to a symbol, in this case a numeric symbol.

The Linker can write a table in its map file showing the line numbers and the final segment numbers and offsets to which they correspond. Some debuggers can

use this information; for instance the Microsoft SymDeb product under MS-DOS took the line-number table from a map file and processed it for use during debugging. The Linker can also write the line number table in an undocumented format into the .EXE file, where CodeView can use it.

Other debugging information is carried in specially-named segments. Two of these that can be observed in C object modules are listed at the end of Table 3.1. It is unknown whether those segment names and classes receive special treatment from the Linker, or what their significance may be to CodeView.

3.3 Controlling the Linker

The Link command is controlled by three kinds of input: its command line parameters, the object files it reads, and a definition file of additional parameters.

3.3.1 Choosing MS-DOS or OS/2 Output

The Linker can link its input to execute in either of two incompatible environments: Real mode or Protect mode. It chooses which form to use based on its input, and it usually picks the correct one. If the Linker resolves an external reference to a dynamic link entry point, or if it is given a definition file, it produces a Protect-mode executable file. If none of these things occurs during the link, the output file will be linked for Real mode.

It is hard to imagine a program for Protect mode that does not include at least one external reference resolved by reference to *DOSCALLS.LIB*, so it is quite unlikely that a Protect mode program will be linked to Real mode format by mistake. The error is more likely to go the other way, when a program written for MS-DOS comes out in the Protect-mode form that MS-DOS can't load.

Reference: executable file formats 4.1

3.3.2 Command-Line Parameters

The Link command takes five primary parameters. It normally takes them as command line parameters, but it can read them in three different ways. The parameters and the rules for entering them are summarized in Table 3.2.

C programmers will normally invoke the linker automatically, as a byproduct of the *CL* command. Those using other languages often, and even C programmers occasionally, have to invoke the Link command directly.

The most reassuring way to enter Link parameters, and the way that places the least burden on your memory, is to let the program prompt you for the parameters it needs. Simply type as many parameters as you wish or can remember, then press Enter. The program's prompts will remind you of what else it needs to know.

The simplest way of entering the parameters (if you are accustomed to Link and feel at home with a command-line interface) is to enter them all on the command line. A one-module program can be linked quite simply with a line like:

```
link theprog,theprog/co/li;
```

or perhaps a line like:

```
link theprog,,,,theprog.def
```

Commands like these look bizarre to some. They certainly prove the adaptability of the human brain, for with only a little practice they make perfect sense and can be rattled off at speed. They can be used to impress the gullible with your technical expertise.

Since two of the parameters may be lists of files, and any of them may take path-strings as well as filenames, the command line can become unwieldy very quickly. When it becomes too long for comfort you may use prompted entry, since with the program prompting, you can break a long list of object files and libraries across multiple lines. Enter one or more (paths and) filenames, ending the line with a plus-sign. The program will issue the same prompt again and you may continue the list.

This quickly gets tedious. The second or third time you have to enter the same long sequence of parameters and options, construct a *response file* instead. A Link response file is a text file containing exactly the lines you would have typed in response to the program's prompts. Instead of typing them over and over to link, type them just once to Copy. Enter:

```
copy con theprog.rsp
```

Table 3.2
The Command Parameters of LINK

LINK *object, output, map, lib, def*

For *object* —	write the (paths and) filenames of one or more object files separated by spaces or plus-signs.
For *out* —	write the (path and) filename of the output file. The default is the filename of the first object file. The file extension defaults to .EXE for a program or .DLL for a dynamic link package.
For *map* —	write the (path and) filename of the link map file. The default is the filename of the output file. The file extension is .MAP.
For *lib* —	write the (paths and) filenames of one or more object libraries separated by spaces or plus-signs. Where no path is given the Linker searches the list of paths set in the environment variable named LIB.
For *def* —	write the (path and) filename of a module definition file. The file extension defaults to .DEF.

End the line at any point and the program will prompt for missing parameters. Write a semicolon at any point to take the default for all remaining parameters. Write *@filespec* at any point to have remaining parameters read from that response file.

and then type exactly the lines you would have typed in response to the prompts of Link. Press F6 and Enter to close the file. From then on run the whole complicated link with the simple command:

```
link @theprog.rsp
```

or whatever the name of the response file may be. This is a great time saver when you are compiling and testing one module of a complicated program again and again.

3.3.2.1 The Input Object Files

The first parameter is a list of the names of the object files to be linked. Every independent object module in the program must be listed, with its path if it isn't in the current directory. The Linker will assume a file extension of *.OBJ*, so that needn't be entered. The first filename becomes the default filename for the output and map files.

When the number of object modules passes two, the Link command becomes distinctly unwieldy to type, and you will want either to prepare a response file or to collect your object modules in an object library that can be named in the fourth parameter.

3.3.2.2 The Output Executable File

The second parameter names the destination of the linked program. The Linker will assume a file extension of .EXE or .DLL as appropriate. Remember that the executable files of running programs and active dynamic link libraries are open files; they cannot be erased or replaced. In order to link a new version of a running program or library, you must specify a path or a filename that is different from the active program.

Reference: loaded programs are open files 4.3.1.2

3.3.2.3 The Link Map File

A link map is a list of the segments and groups in the output file, with their ordinal segment numbers and sizes. It also shows the segment number and offset of the program entry point.

There are two things that will make the Linker write a link map. One is the entry of a (path and) filename in the third parameter position (a file extension of .MAP is supplied by the Linker). The other is the use of the command option */MAP*. If the parameter is omitted and the option is not used, no map file is written. If the parameter is omitted but the option is used, the map file is written using the filename of the output file.

Reference: map option 3.3.3.2

3.3.2.4 Choosing Libraries to Search

The fourth command parameter is a list of the (paths and) filenames of object libraries that are to be searched. The Linker will assume a filetype of .LIB.

When linking compiled programs it is often unnecessary to name any libraries at all. The compiler will usually insert object records that give the filenames of the libraries that contain that compiler's runtime support modules. These libraries will be searched automatically (unless the */NODefault* option is used). Use the parameter to list libraries of object modules that you've created, or toolkit libraries from third parties, or custom versions of the compiler libraries you've prepared.

It is usually not necessary to give paths for libraries, since the Linker will search for the filenames along the paths specified by the *LIB* environment variable. Make sure that its value is a list of paths (similar to the lists in the *PATH* or *DPATH* environment variables) on which object libraries can be found; then give just the filenames of the libraries.

Reference: no-default-library option 3.3.3.4

3.3.2.5 The Definition File

A definition file can contain a variety of Linker control statements, and these are discussed in a topic of their own below. A definition file is not required for most programs.

Reference: definition files 3.3.4

3.3.3 The Command Options

The Linker accepts a variety of options. Each option is a word that is prefixed with an initial slash to distinguish it from a filename. The options that are relevant to Protect-mode programs are summarized in Table 3.3. The words may be abbreviated to one or a few characters. In that table the minimum abbreviations are capitalized.

3.3.3.1 Command Option Syntax

The Linker is flexible about the syntax of its options. Any of the options may be entered as part of any of the five command-line parameters. They may be inserted before filenames or after them. They may be set off by spaces or closed up to other options or filenames.

In a response file it is especially convenient to set out all the options on a single line for visibility and ease of editing. The option line may be any line of the response file. So long as it ends with a plus sign, the Linker will consider it as part of whatever line follows it. The first line of a response file might be:

```
/MAP /LINENUMBERS /CODEVIEW /STACK:4096 +
```

The trailing plus sign makes the line part of the line that follows.

Reference: response file 3.3.2

3.3.3.2 The Information Options /HE, /IN, /MA

Three options cause the Linker to display information about its operations. The */HElp* option produces nothing but a list of all the option words on standard output without explanation or indication of their abbreviations.

The */INformation* option causes the Linker to write a trace of its operations on the standard error file, separate from the standard output display and from the map file. The trace consists primarily of a list of the object modules that the Linker processes, in the order it reads them, including the full path and filename of the library from which the modules come.

This trace can be useful in two cases. Sometimes there is a question of which version of a module is being included, or which copy of a library is being searched. Other times there is a question as to which included module provokes a Linker error message. When standard output and standard error are the same file, error messages appear just after the name of the module that was being processed when the error was discovered.

Table 3.3
Summary of Link Options for OS/2

Information	
/HElp	Display a reminder of all options and stop.
/Information	Display detailed trace to stderr file.
/Map	List public symbols in link map file.
Handling of Object Files	
/NOIgnorecase	Distinguish lowercase from uppercase names (required for C language modules, harmful with assembler or Pascal).
/NOExtendeddictsearch	Do not resolve artificial externals defined by Linker.
/NODefaultlibrarysearch	Do not search libraries specified object module records.
Debugging Information	
/LInenumbers	List addresses of source line numbers in map file.
/COdeview	Include symbolic information and the line number table in the executable file for use by CodeView.
Handling of Output File	
/Alignment:*number*	Align segment images in file on *number* boundaries for speed in loading; default is 512, the usual sector size.
/DOsseg /NONulldosseg	Write segments in standard order in the DGROUP. /DOsseg inserts 16 bytes of zeros at the start of the first code segment, /NONull does not.
/Farcalltranslation	Request conversion of redundant far calls to the faster sequence push cs; call near; no-op.
/PACkcode:*maxsize*	Limit packing of adjacent segments into physical segments to physical segments of *maxsize* (default 65536).
/NOPackcode	Disable packing of adjacent segments entirely.
/STack:*number*	Force stack space in DGROUP to *number* bytes.

The */Map* option causes the Linker to write a link map file regardless of whether one was specified in its third command parameter. In addition it causes the

Linker to put more information into the map. Besides a list of segments and groups, it also lists all public names that were defined. These are listed twice, once in sequence by name, and once in sequence by segment number and offset.

The */Map* list of public names is a useful debugging tool in MS-DOS. There, the far address of a public name can be calculated by adding the program's segment origin to the segment address in the link map; after which a public object can be found even with a simple tool like MS-DOS Debug. Since the same information and a great deal more is available online under CodeView, the */Map* list is less important in OS/2.

3.3.3.3 Letter Case of Names and Option /NOI

Recall that much of what the Linker does is based on matching character strings for equality. It compares segment names and classes when combining segments, matches external and communal names to public names to resolve bindings, and compares segment names to the lists of segments in group definitions.

By default the Linker ignores letter case when making these comparisons. That's the right behavior when it is dealing with names defined by the Assembler or a FORTRAN, BASIC, or Pascal compiler. Those languages also do not distinguish letter case in names. Usually, but not certainly, such a language processor will force names to uppercase when it writes an object file. It remains possible that an external or public name might be defined as *PubProc* in one object file, but the matching public or external name will be *PUBPROC* in its object file. The user will expect the Linker to match the names, and it will by default.

C programs, and possibly other languages, distinguish between lowercase and uppercase letters. When linking C programs, it would be quite wrong to match *PubProc* to *pubproc* or *PUBPROC*; those are distinct names in C. If the Linker treated them as equal, the wrong ones might be matched (more likely, there'd be an error message about the name being defined more than once).

When linking a C program, the */NOIgnorecase* option must be given. (The *CL* command that is commonly used to compile and link C programs includes */NOI* automatically in the parameters it passes to the Linker.)

What can be done to link a C program with a module from a language that ignores case? The simplest solution is to declare the extern in C using capital letters, and then to make sure that other language processor emits the name in uppercase. The Assembler supports an option */MU* that causes it to force all

names to uppercase. The Microsoft Pascal compiler writes only uppercase names in its output. If there is any doubt that another compiler might not force names to uppercase in the object file, the programmer should take pains to write all public names in uppercase. (A compiler might not force names to uppercase, but surely it would never force an uppercase name to lowercase!)

The names of OS/2 system functions follow the Pascal convention; they are written all-cap in the *DOSCALLS* object library. They can be written mixed-case in an assembly or Pascal source file, but must be declared all-cap in a C source file.

There is a C language convention that all-cap names are only used with names defined for macro replacement. All-cap external names conflict with this usage; besides, they don't look right in the context of a C source file with its lowercase variables and keywords. Worse, the functions supplied by the Presentation Manager are defined in mixed-case letters. One solution is to declare the extern in uppercase, and then to use the C macro facility to define a lowercase or mixed-case synonym. That's what's been done in the C header files in the Disk Appendix to this book, for example:

```
extern void far pascal DOSSLEEP(long);
#define DosSleep DOSSLEEP
```

The function may be called as *DosSleep* and the C preprocessor will convert it to uppercase.

3.3.3.4 Skipping Default Libraries with /NOD

Compilers usually write into object files the filenames of object libraries of run-time support code. When the Linker sees these object records it notes the file-names. After it has searched any libraries that were given in its fourth parameter it searches these object libraries. Even if it doesn't need them, it searches for the library files and will issue a warning message if it can't find them.

There are often good reasons for averting the use of the default libraries named by the compiler. They may have been renamed or split up, or there may be a customized local version of them. Even though the libraries named in the command parameter satisfy all external names, the Linker will still protest if it can't find the default libraries. The Linker can be made to ignore them completely with the */NODefaultlibrary* parameter. When this is given, all the libraries needed must be named in the command line or response file.

3.3.3.5 Including Debug Data with /LIne and /COde

The */Line* option causes the Linker to write a table into its map file showing the segment number and offset corresponding to each source file line number. The Microsoft SymDeb debugger, and perhaps others, can use this information. The */COdeView* option causes it to insert more extensive debugging information into the executable file itself, where the CodeView debugger can find and use it.

3.3.3.6 Ordering DGROUP With /DOsseg and /NONull

In Real-mode programs the */DOsseg* option specifies a particular order for all segments. In Protect mode, the order of physical segments is not significant, since segments are known by selector numbers that have no arithmetic relationship. However, the */DOsseg* option also causes a particular order of segments within the group, *DGROUP*. This order may be relied upon by compilers.

The */DOsseg* option causes the Linker to insert a paragraph (16 bytes) of zero bytes at the start of the first code segment. This padding is not required in Protect mode. The */NONull* option calls for the ordering of segments within *DGROUP* but does not insert the padding bytes in the first code segment.

Reference: DGROUP 3.2.4.5

3.3.3.7 Setting File Alignment With /Alignment

The binary image of each physical segment is written separately to the output file. By default the Linker starts each sector on a 512-byte multiple, so that it will begin on a new disk sector. This yields a slight improvement in the speed of loading the program; more important it may slightly improve the speed with which a code sector can be reloaded after it is swapped out.

The */Alignment* option takes a number that must be an integral power of 2; this number is used as the file alignment boundary for sector images. If the disk where the program is stored has a physical sector size greater than 512 bytes, the option can be used to force sector images to start on these larger sector boundaries. On the other hand, if a program consisted of many small segments, some disk space could be conserved by specifying a smaller alignment.

Do not confuse this alignment with the storage alignment specified by a segment's Align type. Physical segments are page-aligned when loaded into storage. This option only affects the layout of the executable file on disk.

3.3.3.8 Packing Code With /NOP, /PAC, /FAR

The */NOPackcode* option prevents the Linker from packing together code segments. It does not affect the combining of segments based on name and class or the creation of groups. The */PACkcode* option can be used to set a maximum size for packed segments; specifying a small number would have much the same effect as */NOP*.

The conversion of far calls into near calls in combined, grouped or packed code segments has been discussed. By default it is not done; the */Farcalltranslation* option enables it.

References: packing segments 3.2.3.5: far call translation 3.2.3.2

3.3.3.9 Setting Stack Space with /STA

Ordinarily, the amount of stack space reserved in the *DGROUP* is established by the aggregate size of the defined segments that have combine-type *STACK* and are grouped in *DGROUP*. The Linker puts them together at the end of the *DGROUP* and will set the SP register to their top offset at run time.

Ordinarily, there is only one defined stack segment, defined by some runtime support module that all programs link. If there's a need for a bigger stack (to support a highly recursive program, for example) there is no convenient way to change the size of that segment and no easy way in a high level language to define additional stack segments.

The Linker comes to the rescue with the */STack* option. The number it specifies replaces the aggregate size of stack segments in *DGROUP* and thus the initial setting of the *SP* register when the program starts. Notice that the */STack* number does not *augment* the stack size, but *replaces* it. It can be used to make the stack smaller as well as larger. The same effects can be had with the *STACKSIZE* statement in a definitions file.

Keep in mind when making the stack larger that, in the Small and Medium models, the *DGROUP* must contain all global variables and the stack and the heap, and that its total size can't exceed 64K bytes.

References: DGROUP 3.2.4.5; memory models 2.3

3.3.3.10 The MS-DOS-Only Linker Options

The OS/2 Linker can create MS-DOS executable files, so it supports options that are applicable only to the MS-DOS environment. These are summarized in Table 3.4. Since they are not relevant to Protect-mode programs they are not discussed further here.

Table 3.4
Summary of Link Options for Real Mode

/Pause	Pause before writing output file
/Exepack	Pack runs of zeros in the Real-mode .EXE file
/CParmaxalloc:*number*	Set maximum Real-mode heap space to 16**number* bytes
/SEgments:*number*	Set maximum number of segments (limit LINK storage use)
/Overlayinterrupt:*number*	Set interrupt number of resident overlay handler
/DSallocate	Load initialized data as high as possible in DGROUP
/HIgh	Tell the Real-mode loader to load the program as high as possible in storage
/NOGroupassociation	Ignore group definitions for compatibility with old systems

3.3.4 The Definition File

A definition file is a text file containing more information for the Linker. One can be named as the fifth parameter of the Link command. The statements that may appear in a definition file are summarized in Table 3.5.

Table 3.5
Summary of Definition File Statements

`NAME` *name* `[windows?]`
Specify program, not dynamic link package, and set *name by* which program will be known in table of loaded programs. For *windows* state the program's relation to Windows:
WINDOWAPI if it must run in the Windows screen group
WINDOWCOMPAT if it may run under Windows or alone
NOTWINDOW if cannot run in the Windows screen group

`LIBRARY` *name* `[init]`
Specify dynamic link package and set name by which package will be known in table of loaded programs. For *init* write the rule for initializing the package:
NITGLOBAL if it should be called once on loading
INITINSTANCE if it should be called for each new client

`DESCRIPTION '`*descriptive line*`'`
Specify a line of text to be included in the .EXE file.

`PROTMODE`
Specify that the .EXE file will never need to be loaded in Real mode (contradicts STUB, REALMODE).

`REALMODE`
Specify that the .EXE file will never need to be loaded in Protect mode (contradicts STUB, PROTMODE).

`STUB` *filespec*
Specify that the .EXE file may need to be loaded in Real mode, and that the MS-DOS executable program *filespec* should be inserted at the head of the .EXE file to get control in Real mode (contradicts PROTMODE, REALMODE).

`STACKSIZE` *number*
Set the required amount of stack space in the DGROUP segment to *number* bytes.

`HEAPSIZE` *number*
Set required free space above stack in the DGROUP segment to *number* bytes.

`CODE` *loading usage priv conf*
Set options for all code segments not specifically listed in a SEGMENTS statement (default given first in each pair):
- For *loading* write PRELOAD or LOADONCALL
- For *usage* write EXECUTEREAD or EXECUTEONLY
- For *priv* write NOIOPL or IOPL
- For *conf* write NONCONFORMING or CONFORMING

DATA *dgroup loading usage priv*

Set options for DGROUP and other data segments not listed in a SEGMENTS statement (default given first in each pair):

For *dgroup* write MULTIPLE or SINGLE or NONE
For *loading* write PRELOAD or LOADONCALL
For *usage* write READONLY or READWRITE
For *priv* write NOIOPL or IOPL

SEGMENTS

Introduce list of segments; order of list is order of segments in output file. List each segment as:

name [CLASS '*class*'] [*minsize*] *options...*

For *name* write the segment name. Enclose the name in quotes if it is DATA, CODE, or another keyword in this table.

write CLASS '*class*' if the segment's class is not CODE.

To load the segment into a space larger than its size as linked, write *minsize* as the necessary size in bytes.

For *options...* write any selection from this list:

- LOADONCALL or PRELOAD
- READWRITE or READONLY (non-CODE segments only)
- EXECUTEREAD or EXECUTEONLY (CODE segments only)
- NOIOPL or IOPL
- NONCONFORMING or CONFORMING

If no option is written, the segment defaults are those set by the CODE or DATA statement. When any option is written, defaults for this segment become those listed first here.

IMPORTS

Introduce list of dynamic link procedures not resolved by an object library. List each as

[*extname*=]*module.entry*

For *module* write the name of the package being imported.

For *entry* write the name or ordinal number of the function that is exported by *module*. If *entry* does not match an external reference, write *extname*= to show the name to be bound to this reference.

EXPORTS

Introduce list of names exported by a dynamic link library (LIBRARY statement used) or public entries to an IOPL code segment. List each exported name as:

pubname[=*name*] [@*number* [RESIDENTNAME]] [NODATA] [*words*]

For *pubname* write the name that callers will give as an external name; it may not be a number or a Linker keyword.

If *pubname* is not a public name, write =*name* to specify the declared public name of the procedure.

For a dynamic link entry imported by number, write @*number*. Add RESIDENTNAME if clients may also link by name to *pubname*.

Write NODATA if a dynamic link package has a DGROUP segment and DATA SINGLE was specified and this procedure does not depend on the DGROUP.

If the exported name resides in an IOPL code segment, write as *words* the number of parameter words (0 to 31) that will be copied to the privilege-2 stack during a call.

3.3.4.1 The NAME and LIBRARY Statements

The *NAME* statement specifies that a program is being linked, while a *LIBRARY* statement specifies that a dynamic link package is being linked. It is when *LIBRARY* is given that the file extension of the Linker's output file is set to .DLL instead of .EXE.

Either statement specifies the name by which a program or package is to be known in the OS/2 kernel's table of loaded programs. The default is the filename of the program file. The significance of this name is discussed with the Loader.

The *NAME* statement includes an optional keyword that indicates whether the program can operate under Windows, that is the Presentation Manager.

The *LIBRARY* statement includes an optional keyword that indicates when the library should be called to initialize itself. By default, a library is allowed to initialize itself just once, when it is first loaded. If *INITINSTANCE* is specified, the library's initialization entry point will be called every time a new client program is loaded and linked to the library.

Reference: loading dynamic link packages 4.2

3.3.4.2 The PROTMODE, REALMODE, and STUB Statements

These statements apply only to programs, not to dynamic link packages. *PROTMODE* is the default; it specifies that a program is never to be loaded in Real mode. An attempt to run the program under MS-DOS will produce an error message of insufficient memory. This is a result of deliberately false data left in the header of the .EXE file by the Linker.

The *REALMODE* statement specifies that the program will only be used in Windows 2.0 in Real mode; it contradicts *PROTMODE* and allows the use of

some keyword parameters of the *CODE*, *DATA* and *SEGMENTS* statements that are not discussed here because they apply only to Real-mode Windows programs.

The *STUB* statement causes the Linker to include the body of a Real-mode program at the head of the .EXE file. If the program is loaded under MS-DOS, that program will get control instead (or first). This is the basic capability that makes the "Family API" workable.

References: bimodal executable files 4.1; heap size in executable file 4.1.2.2.1

3.3.4.3 The STACKSIZE and HEAPSIZE Statements

The *STACKSIZE* statement does just what the */STACK* command-line option does: it establishes the amount of *DGROUP* space that will be devoted to stack space in the loaded program.

The *HEAPSIZE* does the same for the heap, which is the portion of *DGROUP* that lies above (at higher offsets from) the stack space. The *DGROUP* segment will be loaded into a larger physical segment space in order to create this space. Large-model programs may or may not use this *DGROUP* heap space; it is probably relevant only to programs that use the Small or Medium models.

The runtime support code for Microsoft C (Small and Medium models) and Microsoft Pascal does not need heap space pre-allocated in this way. When heap space is needed it extends the *DGROUP* segment in place using OS/2 services. Other language implementations might not know how to do this and so might need pre-allocated heap space.

References: DGROUP 3.2.4.5; stack option 3.3.3.9; memory models 2.3

3.3.4.4 The CODE and DATA Statements

The Linker notes certain attributes for each segment of code and data in the .EXE file (the uses of these are discussed in more detail in the Loader topic below). The *CODE* and *DATA* statements of a definition file set the default attributes for code and data segments, respectively. Most of the defaults are chosen from pairs of alternates.

The choices to be made are summarized in Table 3.5. The specific keywords and concepts are discussed at greater length under the *SEGMENTS* statement later. The unmodified defaults shown in the table are suitable for almost all situations.

Usually only one or a few segments need different treatment and these can be described using the *SEGMENTS* list.

3.3.4.4.1 The DGROUP Choices

One keyword in the *DATA* statement is not a default value for all data segments but rather tells the Linker and Loader how to handle the *DGROUP* segment when one exists.

The *DGROUP* contains all the miscellaneous static variables for code generated by a compiler or linked from a compiler's runtime support library. For Small and Medium model programs, it contains the stack and heap as well. The code that uses *DGROUP* generally assumes exclusive use of it and can't cope with sharing the segment. Only in very unusual circumstances would a *DGROUP* be shared across multiple processes.

The *MULTIPLE* option specifies the normal situation: that the *DGROUP* segment is unshared and a unique copy should be created for every instance of this program. The *SINGLE* option specifies that a single copy of the *DGROUP* is to be created and shared by every instance of this program. The *NONE* option specifies that no *DGROUP* is defined in the program.

References: DGROUP 3.2.4.5; memory models 2.3

3.3.4.5 The SEGMENTS List

The word *SEGMENTS* introduces a list of segments that are to receive special treatment when they are loaded. Its syntax is summarized in Table 3.5. The segments in the list are identified by name, with the name qualified by class-name if necessary. Segments named in the *SEGMENTS* list are not physically packed with other segments, regardless of the */PACkcode* option.

One reason for listing a segment here is to force it to be loaded at a certain size greater than its size as linked. The *STACKSIZE* and *HEAPSIZE* statements can be used to force up the size of the *DGROUP*; the *minsize* parameter can force a size for any segment.

The other reason for listing a segment is to give it attributes different from the default attributes set by the *CODE* or *DATA* statement. Notice that if any at-

tribute is specified here, the defaults for the others change. As a rule, if one attribute is listed, all should be listed.

Reference: packing physical segments 3.2.3.5

3.3.4.5.1 PRELOAD and LOADONCALL

When the Loader first loads a program or dynamic link package, it immediately reads into storage all segments that are marked *PRELOAD*. Most segments should have this attribute, since most segments will be accessed within a fairly short time of loading.

The *PRELOAD* attribute is effective only at load time; it does not retain a segment in storage. Any segment may be swapped out eventually, if swapping is active and if the segment goes unused for long while the system is short of space.

LOADONCALL is the opposite of *PRELOAD*. When the Loader first loads a program or dynamic link package, it assigns selector numbers for *LOADONCALL* segments but does not allocate space for them or read them from disk. The Present bit is set false in the descriptors for these selectors. If and when the program attempts to refer to one, a hardware trap will notify OS/2 that the segment is needed, and it will be loaded at that time.

The point of the *LOADONCALL* attribute is to save time and space by omitting to load segments that are rarely needed. The program will start execution sooner when a bulky part can be left behind on disk until it is needed.

A program has to be designed to take advantage of the attribute. Low-use data has to be strictly segregated into the segment. Loading is an all-or-nothing process; if the program refers to even a one-byte variable in the segment, the whole segment will be loaded. Furthermore, the *LOADONCALL* segments should be large; there's not much point in deferring the load of a few kilobytes. Bulky help displays are good candidates, since the user might never ask for help (and experienced users are also the ones who appreciate snappy response time).

References: segment descriptors 2.4.3; present bit 2.4.3.3.1

3.3.4.5.2 READONLY and READWRITE

When a data segment is entered into the Local Descriptor Table, it can be marked read-only or read-write. Data segments are ordinarily read-write since they

ordinarily contain variables that the program will store into. *READWRITE* is the default attribute for data segments, unless it is changed with the *DATA* statement.

If a segment contains only data that is read and never changed, the segment could be marked *READONLY*. There are two good reasons for doing this. First, it sets a hardware guardian on the segment so that if a program tries to store into it by mistake, the error will be caught at once. Second, it makes it safe to mark the data segment *SHARED* as well: since modification of the data will be physically impossible, it must be safe to share it.

References: load descriptor table 2.4.4.2; read-only bit in descriptor 2.4.3.3.2

3.3.4.5.3 EXECUTEONLY and EXECUTEREAD

When a code segment is entered into the Local Descriptor Table, it can be marked as execute-only or readable. The contents of an execute-only segment may be fetched as machine instructions only. Access to it by any other means causes a hardware trap. For example, it would not be possible to copy a literal character string or a constant floating-point number from the code segment to initialize a variable. Since this kind of usage is common at least in assembly programs, code segments are by default *EXECUTEREAD*, that is, readable as data.

One reason for marking a code segment *EXECUTEONLY* is to protect the contents of a proprietary program against theft. *EXECUTEONLY* code can be executed, but it can't be copied. This would most commonly be a consideration with dynamic link packages. (On the other hand, how can you protect the disk file against copying?)

References: local descriptor table 2.4.4.2; execute-only bit in descriptor 2.4.3.3.2

3.3.4.5.4 IOPL and CONFORMING

When a segment is entered into the Local Descriptor Table its descriptor contains a code privilege level. The ordinary privilege level of any segment outside of the OS/2 kernel is 3, the most humble. This privilege level is signified by the default attribute *NOIOPL*.

The *IOPL* attribute on a segment causes the Loader to give it privilege level 2, the level at which machine instructions related to I/O can be executed. If a code segment contains code to operate on hardware devices (for instance, code to write

to the control registers of the video display adapter) it must be given the *IOPL* attribute.

When a data segment has I/O privilege, it can *only* be accessed from code that is operating with I/O privilege. Ordinarily a data segment would not be given the *IOPL* attribute, since code running with I/O privilege may access data that has the lesser, *NOIOPL*, level.

When a code segment has I/O privilege, its entry points may *not* be called by unprivileged code in the normal way. Instead, a call gate must be built for each entry that is to be called from outside the segment. (Calls from one procedure to another within the segment may be made in the normal way.) The *EXPORTS* list in the definition file is the method by which you specify the callable entry points in an *IOPL* code segment.

Ordinarily, code running at the normal privilege level may not jump or call into a segment that has *IOPL* privilege. The CPU hardware will not permit it, and will trap out to OS/2 if it is attempted. There is an exception allowed, however. If a code segment is marked with the *CONFORMING* attribute, it may be entered from code at a humbler privilege level without penalty. However, the humble privilege level is retained; the conforming code may not execute any instructions that its caller could not execute.

Apply the *CONFORMING* and *IOPL* attributes together to any code segment that contains public entry points that may be used at the normal, *NOIOPL* privilege. These entry points don't need call gates, so they needn't be mentioned in the *EXPORTS* statement. They may not execute I/O instructions, either. Public procedures that execute privileged instructions must be entered through a call gate in order to accomplish the jump from normal to *IOPL* privilege. In general it is best to use an *IOPL* segment only for code that needs I/O privilege.

> **References:** local descriptor table 2.4.4.2; privilege level in descritpr 2.4.3.3.3; conforming bit in descriptor 2.5.2; call gates 2.5.1

3.3.4.6 The IMPORTS Statement

Ordinarily all of a program's external references are resolved to public names in other object modules. When the external references are to procedures in a dynamic link package, the dynamic link package will ordinarily be represented by an object library prepared with the Implib utility. The records in that artificial ob-

ject library tell the Linker that a public name is defined by a dynamic link package and will be resolved by the Loader.

The *IMPORTS* statement can also be used to resolve external references to dynamic link entry points. It introduces a list of public names that will be available at load time in dynamic link packages. The Linker handles these the same way it handles the names prepared by Implib: it uses them to resolve external names used in the program, and records them in the .EXE file to be resolved by the Loader.

One minor advantage that *IMPORTS* has over an object library created by Implib is that the statement allows an alias name to be used for the dynamic link procedure. An external reference to one name, written into the program, can be resolved to a dynamic link function that has a different name.

Reference: table of imported names 4.1.2.2.4

3.3.4.7 The EXPORTS Statement

The *EXPORTS* statement introduces a list of public procedures. This list has two distinct purposes, depending on whether a program is being linked or a dynamic link package (whether the *NAME* or *LIBRARY* statement was used). For a program, an *EXPORTS* list is needed only to define public entry points in I/O privileged segments. For a dynamic link package, the list defines the routines that the package makes available to other programs.

Reference: entry table in executable file 4.1.2.2.3

3.3.4.7.1 Entries to IOPL Code

Public entry points to a code segment with I/O privilege level may only be called if they are listed in the *EXPORTS* list. For this purpose the form of each list entry may be simplified to:

```
pubname  words
```

where *pubname* is the name of a public entry point in an *IOPL* segment and *words* is the number of stacked parameter words that the procedure expects. This supplies the Linker and Loader with enough information to build a Call Gate descriptor for the procedure.

3.3.4.7.2 Dynamic Link Procedures

When a dynamic link package is being linked, the *EXPORTS* list documents the procedures that the package makes available to its clients. In the usual case, the exported entry point is declared in the source code as a public name, and the programs that use it will declare the same name as an external name. In this case the form of a list entry may be simplified to:

```
pubname
```

that is, just the name of the procedure. The listed public names are incorporated in the executable file and are retained in storage when the package is loaded.

In some cases a dynamic link package does not export its services by name but by number. In this case the form of a list entry becomes:

```
pubname @number
```

where *pubname* is the public procedure that is to be known as ordinal *number*. Both name and number will be retained in an object library created by Implib so clients that link through an object library may still refer to the procedure by name. When the link is via an *IMPORTS* list, the number will have to be specified correctly. It is possible for a client program to link at run time using the *DosGetProcAddr* call. If the public name is to be retained in storage so that it can be found at run time, the keyword *RESIDENTNAME* must be appended.

In either case it is possible to make the *pubname* an alias for a different public name, so that the name linked by the client is different from the declared entry name. There are a number of uses for this feature. One is to direct calls to unimplemented features to a single error handler.

The keyword *NODATA* is only coded when the same definition file specifies *DATA SINGLE* for the *DGROUP*, and then only when the first instruction at the entry point *pubname* isn't:

```
mov ax,seg DGROUP
```

Ordinarily dynamic link packages either have no *DGROUP* at all (*DATA NONE*, the usual case when assembly language is used to write the package) or else they require a new *DGROUP* for each linked client process (*DATA MULTIPLE*, the usual case when compiler-generated code is being bent to fit the needs of a dynamic link package). In the remaining case—the package uses a *DGROUP* and

shares it among all instances of the package—the first thing a procedure normally does is load the selector for *DGROUP* into *AX* and thence into *DS*. This won't be the case in a procedure that doesn't rely on *DGROUP*, and the *NODATA* keyword is required to convince OS/2 that you really knew what you were doing.

A count of parameter words must be coded if the exported service resides in an *IOPL* code segment.

Reference: run-time linking 4.5; *DosGetProcAddr* 4.5.4

3.4 Summary

The Linker accepts such a wide variety of inputs because it has to deal with a huge variety of program forms. It is reassuring to remember a large number of programs can be linked successfully with a command like:

```
link myprog;
```

using defaults for everything. An even larger number of programs can be linked with the slightly more complicated form that specifies a couple of modules, a couple of libraries, and an option or two:

```
link first second,,/map/co,onelib another;
```

Only the most sophisticated of applications ever requires a definition file.

Dynamic link packages require a definition file, but that can be written once and then all but its name forgotten. The Disk Appendix to this book includes a dynalink package named *GLOBENV*, and it is linked with the simple command:

```
link globenv,,,doscalls,globenv
```

So although the discussion of the Linker's inputs and options has extended over many pages, in use it is not an overwhelming program.

4

Loading

The final result of the complicated work of the Linker is an executable file. If it represents a program, its file extension will be .EXE; if it represents a dynamic link package, the extension will be .DLL. The internal form of the files is almost the same. Each represents a collection of segments, with the supplementary information that will let the Loader resolve the final bindings between them. To describe this file is to describe the process of loading.

4.1 The Executable File

Under MS-DOS an executable file is a fairly simple binary image of the program. In OS/2 the file contains a much richer variety of information—sometimes including an entire MS-DOS executable file.

4.1.1 The MS-DOS Executable File

We will summarize the form of a Real-mode executable file for two reasons: the OS/2 executable file format is based on it and partly compatible with it, and MS-DOS Loading is good background for OS/2 Loading.

4.1.1.1 The Real-Mode Binary Image

An MS-DOS executable file contains a single binary image of the entire program as it will appear in storage. The boundaries between defined segments are not part of this image; it is a single sequence of bytes. In Real mode the hardware does not recognize segment boundaries, so there is no point in retaining them.

The Real-mode binary image is accompanied by a table of fix-up records. A fix-up points to a word in the binary image that contains a segment selector. Recall that in Real mode, a selector is based on a physical address—it's the number of a storage byte divided by 16. Where the program contains a selector, the Linker has left a segment address relative to the beginning of the binary image of the program. These selectors would be correct and ready to use if only the program were loaded beginning at address zero.

The MS-DOS Loader chooses the actual load address of the program and reads the binary image into storage starting at that address. Then it scans the table of fix-ups and fixes up each segment selector in the program by adding the segment selector of the program's load address into it. The fixed-up binary image is ready to run in Real mode.

Reference: real-mode addressing 2.4.1

4.1.1.2 The Real-Mode Header

The initial bytes of an MS-DOS executable file contain supplementary information about the program. One word specifies the size of stack space needed (from the Linker's */Stack* option). Another word specifies the minimum amount of free storage that must remain above the program image after it has been loaded and the stack space allocated (from the Linker's */CParm* option or the *EXEMOD* utility). And there are words that let the Loader prepare the program's initial *CS*, *IP*, *SS*, and *SP*.

Finally, the file header contains the size of the program image. In a program linked for Real mode only, there would be no data in the executable file past the end of the program image. The MS-DOS Loader has no reason to read beyond that point in the file, and it will not.

4.1.2 The OS/2 Executable File

The OS/2 executable file format includes an MS-DOS file header for compatibility, but retains much more information about the program. In the Disk Appendix for this book is a program that dumps the contents of an executable file.

4.1.2.1 Compatibility with MS-DOS

The first bytes of an OS/2 executable file are formatted to the requirements of the MS-DOS loader. If the OS/2 program cannot execute in Real mode, the program-size word will be set so large that MS-DOS will reject the file with the message "Program too big to fit in memory."

At offset 03ch in an MS-DOS program file, there were four reserved bytes. OS/2 has preempted these. The OS/2 Linker puts a 32-bit file pointer to the first byte of the Protect-mode program information into them. If the program is only for Protect mode, the pointer value will be 040h; that is, a pointer to the very next byte of the file.

However, there is nothing to prevent the Protect-mode program from being much further on in the file. The file space between the MS-DOS header and the Protect-mode program is available for a Real-mode program. If there is a Real-mode program, the MS-DOS Loader can load it correctly. It will have no reason to read past the Real-mode image, so the MS-DOS Loader will be unaware of the protect-mode program.

The Protect-mode Loader, on the other hand, uses its file pointer to skip over the Real-mode program data. The first two bytes at that offset should be *NE* (for New Executable). If the file does not contain a Protect-mode program the file pointer will be zero; since the first two bytes of the file will not be *NE*, the OS/2 Loader will give up and issue the message "DOS0191: the system has detected an unacceptable signature in the file."

The outcome of these maneuvers is that a file of type .EXE may be for MS-DOS only, for OS/2 only, or may contain programs for both systems. Both loaders diagnose errors correctly. Neither Loader ever wastes time reading the other's data. This clever design lets the two systems share a file system, and makes the Family API compatibility scheme work.

4.1.2.2 The Module Table Data

The OS/2 part of the file begins with descriptive information, including a table with an entry for every segment. This *module header* is kept in storage by OS/2 so long as the program or package is loaded. The module headers for all loaded programs are grouped in a table called the *module table* in some OS/2 manuals.

4.1.2.2.1 The Module Header

The header begins with global information about the module. Much of this information corresponds directly to Linker options or definition file statements.

The header contains a flag showing whether the module is a program or a dynamic link package. Other flags show whether a *DGROUP* segment exists and if it does, whether it is single or to be created anew for each instance of the program (this is direct from the *DATA* statement in the definition file). The initial heap size is recorded (from the *HEAPSIZE* statement).

For a program, the header notes the segment number and offset of the program's entry point, and the segment number and offset for its initial stack pointer. For a dynamic link package, only the program entry point is noted.

References: DGROUP 3.2.4.5; data statement 3.4.4.1; heapsize statement 3.4.3

4.1.2.2.2 The Segment Table

All of the segments of the module are listed in a compact table. The table shows the size of the binary image of the segment and where it starts in the file, and the size the segment is to have when loaded.

The loaded size may differ from the size of the binary image. The Linker implements the */Stack* option by adjusting the loaded size of the *DGROUP* segment. Common segments and uninitialized data segments may have no binary image at all, but a large allocation size.

A word of flags in the segment table reflects all of the attributes that can be specified in the *CODE*, *DATA*, or *SEGMENTS* statements in the definition file: whether the segment is code or data, whether it is read-only, and so forth. The segment's privilege level is shown as well; normally it is 3, but *IOPL* segments have privilege level 2.

The names and classes of the segments are not included in the file, nor are their align-types or combine-types remembered. All that information was used and discarded at Link time.

References: stack option 3.3.3.9; definition file statements 3.4.4, 3.4.5

4.1.2.2.3 The Entry Table

Another compact table lists all the entry points that were listed in the *EXPORTS* list in the definitions file. The segment number and offset for each entry is listed. If the segment has I/O privilege, the number of parameter words that should be copied to its stack is recorded.

The only entry points that are recorded here are the ones that require attention at load time: public entry points to dynamic link packages because other programs will link to them, and entries to IOPL segments because call gates have to be built for them. The character string names of entry points are not recorded in this table.

References: exported names in definition file 3.3.4.7

4.1.2.2.4 Tables of Names

Some character strings are retained. The Imported Names table contains the character names of dynamic link packages that are to be linked (almost every program will contain at least the string *DOSCALLS* in this table). Where package functions are linked by name instead of ordinal, the entry names are also listed as Imported Names.

The Resident Names table contains at least the name of the program—the name specified in a *NAME* or *LIBRARY* statement in the definition file, or the filename of the executable file by default.

In a dynamic link package, all the public names that were not exported by ordinal, and those that were given as ordinal numbers but also specified as *RESIDENTNAME*, are listed in a table of Resident Names. Each of these name strings is accompanied by an index into the Entry Table, so a reference to the name can be translated into an entry number.

There is also a Nonresident Names table. It is in the executable file but is not retained in storage. Usually it contains only the string specified by a *DESCRIPTION* statement from the definition file.

References: imported names 3.2.1.3.2.5, 3.3.4.6; name/library statements 3.3.4.1; exported names 3.3.4.7

4.1.2.2.5 The Module Reference Table

Almost every program (and most dynamic link packages) will contain dynamic links to the *DOSCALLS* package. There may be links to other packages. The Module Reference table is a list of the names of all dynamic link modules the program links to. It is a compact table because each entry is simply an index of a name in the Imported Names table.

When a module is loaded, the Loader runs through this table and makes sure that all of the needed modules are also loaded. For every entry in the Module Reference Table it searches its Module Table for another module that has that name (in its Resident Names table).

4.1.2.3 The Segments

The binary images of the segments themselves follow all the header information in the file. Each segment's image is aligned in the file on the boundary specified in the Linker's */Align* option—usually 512-byte boundaries, corresponding to the usual disk sector size.

The Loader can use the information in the Segment table to load any segment from disk. After it allocates storage, it can skip to the start of the segment image in the file and read it. Then it may have to perform some fix-ups.

The binary image of a sector may be followed by a series of fix-up records. Each record specifies an offset, a selector, or a complete far pointer within the binary image, and tells what kind of information it needs to make it complete.

Reference: alignment option 3.3.3.7

4.1.2.3.1 Fixing Up Internal References

The most common kind of fix-up is an internal reference; that is, a reference to another segment of the module. The fix-up record gives the logical segment number and the offset. After the Loader has assigned selector numbers to segments it goes back and fixes up the selector numbers in these references.

A special form of internal reference is one that refers to an entry in an IOPL segment. This fix-up record, instead of giving the segment number and offset, con-

tains an index into the Entry table. The Loader will fix up the reference to address the call gate it has built for the exported entry.

Reference: entries to IOPL code 3.3.4.7.1

4.1.2.3.2 Fixing Up Imported References

References to dynamic link packages are reflected in fix-up records. There are two kinds of these: imports by name and imports by ordinal. Either kind contains an index into the Imported Names table where the name of the imported module is recorded.

A fix-up for a procedure imported by ordinal contains the ordinal number of the entry point. The Loader refers to the Entry table of the target module to discover the segment number and offset in that module.

A fix-up for a procedure imported by name contains an index into the client module's Imported Names table. The Loader takes that name string and uses it to search the Resident Names table of the target module. From that it gets the Entry Table index in the target module.

4.2 The Loading Process

Most of the work of Loading is implicit in the contents of the executable file. The first step is to read a 32-bit file pointer from offset 03ch in the file. The file is read from that point, skipping over any MS-DOS executable image that may exist. The first two bytes must be a signature, the letters *NE*.

The Loader reads several hundred bytes of descriptive information into its Module Table. The information includes the Segment table that will let the Loader read each segment individually from the file, and the Module Reference table that lists every dynamic link package this module needs.

The Loader examines the Module Reference Table and checks its table of loaded programs to make sure that every needed dynamic link package is present in storage. If any are not, the Loader turns aside at this point and loads them first before continuing with the present load. The load of a dynamic link procedure is exactly like the load of a program. The format of the executable file is the same, and the process of loading is the same. In essence, the Loader calls itself recursively to load needed packages.

The dynamic link packages that are called may need initialization. When a package was linked with the *LIBRARY INITGLOBAL* statement (or default), the Loader calls the package's program entry point just once, the first time it is loaded. If it was linked with *LIBRARY INITINSTANCE*, the package entry point is called during the loading of any program that names the package.

With all packages loaded, the Loader can choose selector numbers for the segments. The selector for a segment of a dynamic link package, or for any segment that may be shared by more than one process, is chosen from a pool of Local Descriptor Table selectors and reserved to that segment. Selectors for segments that won't be shared between processes are simply taken in sequence up the LDT.

Once selectors have been chosen the Loader can create the segments in storage. For each segment it must allocate storage, read the binary image from disk, and perform any fix-ups. All the information needed to do the fix-ups is available in storage at this point.

When the program image is complete and the LDT built to describe it, the program is called a *process*. A thread is created to animate the process and it begins execution. This is discussed in the chapter on the Execution Environment.

> **References:** library statement 3.3.4.1; selectors for shared segments 2.4.6; process 1.2.3; thread 1.2.4

4.3 The Use of Loaded Modules

Once it has been loaded, a module—that is, a program or a dynamic link package—is kept in storage and in the Module Table so long as at least one process is using it. OS/2 tries hard to avoid reloading a module that has already been loaded.

4.3.1 Instances of Loaded Modules

When a program is started it becomes a process, and its module is noted as having one reference. Ordinarily only one instance of a program is started. When it terminates, the use count of the program module drops to zero and the module is unloaded—its segments are discarded and information about it is dropped from the Module Table.

However, it is possible to start more than one instance of a program. When asked to load a program, OS/2 searches the Module Table first, before it looks for files on the execution path list. If it finds a loaded module with the correct name, it uses it, and never checks the disk for an executable file at all. Each instance of a loaded program increases the use count of the module, which will not be unloaded until they all end.

A dynamic link package will be kept in storage as long as any other loaded module refers to it. Every module contains a Module Reference table, and the use count of every module named in that table is incremented. When a module is unloaded, the use count of every module named in its Module Reference table is decremented.

4.3.1.1 Priority of Modules Over Files

To repeat: the Module Table is searched first, before any disk directory is searched for program files. This is true even if you specify a complete drive and pathname for the program file! The priority of module names over file names can cause unexpected effects. One is that you can never execute a program whose filename is the same as the name of a loaded module.

For proof of this, rename any executable program to have the name *DOSCALLS.EXE*, and try to invoke it. The program will not be loaded. Instead the loader will display the message "DOS0190: the file *DOSCALLS* is not an acceptable module." This is not an accurate statement—you know that the file *DOSCALLS.EXE* is a perfectly acceptable program and you can easily prove it by renaming to some other filename and executing it.

What happened? The Loader searched the Module Table for a module with the name *DOSCALLS*, and found the dynamic link package that represents the OS/2 application program interface. Since it's a package not a program, the error message was issued. The message should say something like "the *module DOSCALLS* is not an acceptable *program*," but no matter. The point is, the disk file was never seen because loaded modules take precedence.

4.3.1.2 Loaded Modules are Open Files

Although a loaded module won't be unloaded, its individual code segments can be discarded if the system runs short of space. When a discarded segment is refer-

enced, the system has to recreate it from disk. That means loading the segment's image and doing the fix-ups again.

Since the system might have to reread part of the executable file at any time, it cannot allow the file to be changed or erased. The executable file kept for input is open so long as the module is resident in storage; it is only closed when the module is unloaded.

This means that the executable file of a loaded program or dynamic link package cannot be erased, replaced, renamed, or opened for output so long as the module is in storage. It isn't possible to link a new version of the .EXE or .DLL file for a loaded module except by using a different filename or directory. If the Linker tries to open the loaded executable file for output, it will cause a file sharing error.

The dynamic link packages that represent the system interface are loaded at all times, and so they can never be replaced! The only way to replace them is to boot the system off of a different drive than normal; then the copies of the system .DLL files on that disk will be in use and the normal ones can be replaced.

4.3 Starting a Process

When loading is done, there is a complete binary image of the program in storage ready to execute. The program segments and the Local Descriptor Table together define a *process*, which is the executable image of a program plus whatever dynamic resources the program owns. The Local and Global Descriptor Tables define the *address space* of the process.

References: descriptor tables 2.4.4; address space 2.4

4.3.1 The First Thread

The Loader creates a single *thread* to animate the process. A thread is an entity that may use the CPU; it consists of a set of registers and the right to execute. (More threads may be created, as discussed in the chapter on Asynchronous Processing.) Threads within a process are assigned Thread ID numbers (TIDs). The first thread has a TID of 1.

This first thread has a special place in the OS/2 scheme. It is this thread that will be redirected into a signal-handling routine when an external signal is received. It

is also this thread that will be redirected to handle Exit List procedures when the process is terminating. (These topics are discussed below.) As a general rule, in a complex application, the first thread should initialize the process, start another thread or threads to do the work, and suspend itself to await events. In an ordinary application, of course, the one thread is all that will exist or needs to exist.

References: creating and managing threads 9.1; signal handlers 5.3.1; exit list 5.3.3

4.3.2 The Initial Registers

The first thread in a process begins with these values in its registers:

- *CS* and *IP* are set to the address of the program entry point as it was defined to the Linker.
- *SS* has the selector for the stack segment or the group of which it is part (usually *DGROUP*), and *SP* has the offset to the top of the stack space in the stack segment.
- *DS* has the selector for the *DGROUP* if one was defined, otherwise it contains zero.
- *CX* has the size of the *DGROUP* if one was defined (a size of 65,536 is represented by zero).
- *AX* has the selector for the environment segment.
- *BX* has the offset in the environment to the first of two strings that define the command line.
- *DI* has the *handle* of the module.
- *ES* contains zero.
- *BP* contains zero.

The environment selector in *AX* and the offset in *BX* can be retrieved later with the *DosGetEnv* call, so it is not crucial to save them at program entry.

The *handle* in register *DI* is not the same as a file handle. It specifies the process's loaded program image within the system's table of loaded modules. It can be used as input to the *DosGetModName* call.

Most of these register items are of interest only if you are writing a program in assembly language. In other languages, the program entry point will be in a runtime support module that will use these values to set up the conditions expected by the generated code. It is an excellent idea to trace through this start-up code under CodeView to see just what initialization is performed. The start-up

code for Microsoft C and Pascal, for example, save values in these public variables in *DGROUP* (all the names begin with two ASCII underscore characters):

- *CX* in *__asizds*
- *SP* in *__atopsp*
- The environment selector from *AX* in *__aenvseg*
- *AX* and *BX* as a far pointer in *__acmdln*

Of course, unless such names are documented in the language manuals they are not official interfaces and might change from release to release.

> **References:** segment registers 2.2.4; DGROUP 3.2.4.5; environment segment and *DosGetEnv* 5.1.5; *DosGetModName* 4.5.3; OS/2 version stored at entry 5.1.1.2

4.3.3 Execution and Termination

The first thread executes the code at the program entry point and this generally initializes the program and calls the code that does the work the program is designed to do. This work might entail acquiring more storage segments, opening files, and creating queues and system semaphores. Actions such as these create resources that are owned by the process. The first thread might also create other threads to execute asynchronously.

OS/2 will retain the process (the program image plus all the resources it has created) in storage as long as there is a single thread to animate it. Threads may terminate themselves by issuing the *DosExit* call. Created threads may also terminate by performing a far return; the system code that implements *DosCreateThread* pushes a return address to a *DosExit* call on the new thread's stack.

When the last thread (which needn't be the first thread) issues *DosExit*, OS/2 disposes of all remaining resources and discards the segments of the program image. OS/2 may have some difficulty disposing of resources, so in general a program should try to clean up after itself. OS/2 does know how to close files that are left open. System semaphores and queues should not be left behind. OS/2 knows how to discard them but it can only give error return codes to the other processes that might have been sharing them.

There are many unexpected events that can terminate a program. A Control-break signal from the keyboard is the most common, but there are also hardware exceptions that are trapped by the CPU, and there is even a system call, *DosKillPro-*

cess, that another process might use. The program can set up handlers to intercept many of these exceptions but eventually something—if only the *DosExit* of the last thread—will lead OS/2 to terminate the process. When everything else is finished, the system reawakens thread number 1 and dispatches it to execute the code of the Exit handlers. When all of these have finished, the program is finally over.

References: *DosCreateThread* 9.1.2; control-break and *DosKillProcess* 5.3.1; exit list 5.3.3

4.4 Initializing Dynamic Link Packages

For the most part, the code of a dynamic link package is only executed when a client process calls one of the procedures in the package. Then the package code is executed by one of the client's threads, with the client's address space of accessible segments (which includes, of course, segments that were linked as part of the package). In this state, the package code has the identity of the client process and thread. A package is like a chameleon, it takes on the color of whichever client is executing its code. The exception is when the package is called to initialize itself.

Dynamic link packages can also be bound by name after the process has been loaded using the *DosLoadModule* call.

The only time that a dynamic link package is executed for its own sake and not as an extension of a client, is when it is initialized. A program entry point may be defined for a package the same way that one is defined for a normal program. This entry point—which need not be one of the entry points the package exports to its clients—will be called by the OS/2 kernel to initialize the package.

The package entry point will be executed by the Loader at one of two times. By default it is executed just once, the first time the package is loaded into storage. However, the Linker can be made to mark the package for per-client initialization. In this case, the package entry point is called by the Loader every time a new client process is created.

When its entry point is called, the register contents are different from those when a program is first entered. The package entry point receives only

- *CS* and *IP* addressing the program entry point
- *SS* and *SP* addressing a usable stack with a return address on it

- *DS* with the selector of the package's *DGROUP* if it defined one, otherwise zero
- *AX* with the package's *handle*

This falls considerably short of what a program receives when it starts up, which is one reason it is usually impractical to write dynamic link packages in high-level languages.

The package entry code is in a sort of process Limbo; it isn't part of any process, strictly speaking, and it is not clear whether it is permitted to do things that require a process ID—things like opening files or system semaphores, or setting signal handlers or exit routines. Anything that takes a process identity as an express or implied parameter should be done on the first call from a client process.

The package entry point must return a nonzero value to its caller in *AX* to indicate successful initialization. If it returns zero to indicate failure, the Loader can't finish loading the client program.

> **References:** *DosLoadModule* 4.5.1; link statement for initialization 3.3.4.1; segment registers 2.2.4; DGROUP 3.2.4.5

4.5 Run-Time Dynamic Linking

Ordinarily a program specifies all its external references to the Linker and all of them are resolved by load time at the latest. However, it is possible to bind a dynamic link package to a process after loading, which the process is running.

4.5.1 Binding a Package with DosLoadModule

The *DosLoadModule* call takes the character-string name of a module. A module in this sense is the executable image of a program or package as loaded by the Loader. The name is the name that would appear in the Loader's table of loaded modules. That's the name given to the Linker in the *LIBRARY* statement in a definition file. It need not be the same as the filename of the executable file!

DosLoadModule looks for the module name in the table of loaded modules. If it doesn't appear, it searches for a file with that name and the extension *.DLL* in the directory where the system dynamic link libraries are stored—the directory whose path is specified in the *libpath* statement of the configuration file. If the library file is found it is loaded, just as it would have been loaded if the current process

had contained a reference to it at load time. If the package uses load-time initialization (the default *INITGLOBAL* option of the *LIBRARY* statement in the Linker's definition file) it is called to initialize itself.

Once the desired module has been loaded, the *DosLoadModule* service checks to see if it has already been bound to the calling process. It might have been; there's no rule against calling *DosLoadModule* with the name of a package that had already been bound by linking, or by a prior call to *DosLoadModule*. After all, one dynamic link package may link to others, so a process cannot really know the full set of packages that it has linked.

At any rate, if the package has not been bound to the process, it is bound at this time. This only means that the segments of the package are entered into the Local Descriptor Table of the process. There cannot be any "fixups"needed to the code of the current process, as there might have been had the package been linked in the usual way. Since the package was not known to the Linker, there cannot be any references to it embedded in the machine code of the process (they would have been *unresolved externs* at link time and no executable file would have been written). If the package uses per-client initialization (the *INITINSTANCE* option of the *LIBRARY* statement) it is called to initialize itself here.

References: library statement 3.3.4.1; table of loaded modules 4.1.2.2

4.5.2 Module Handles

DosLoadModule returns a *handle* that can specify the package it has loaded in other system calls. This is the only way of obtaining a module handle for a package that is not already bound to the current process; it loads the package (if necessary), binds it (if necessary), and then returns a handle for it.

The *DosGetModHandle* service will return the handle of a package that is already loaded, whether it is bound to the current process or not. One way of finding out whether or not a package has been loaded is to pass its name to *DosGetModHandle*. If it returns an error code, the package is not in the OS/2 kernel's table of loaded modules (and definitely not bound to the current process).

If *DosGetModHandle*'s return code is zero, it also returns a handle that stands for the package. In this case the package is in storage, but it is not necessarily bound to the current process.

There's one other way of getting a handle for a module. A package that you write could note its own handle (from register *AX* when it is entered for initialization), save it in a data segment, and return it to a client process on a call to some exported procedure.

All of these handles are *not* taken from the same number set as the handles of files, pipes, or queues, nor are they similar to the handles of system semaphores. From their values they appear to be offsets, probably into some OS/2 kernel table, but that isn't documented. Whatever they mean, they don't reduce the number of available file handles.

Reference: table of loaded modules 4.1.2.2

4.5.3 Module Filespecs

It is possible to find out the complete, fully-qualified filename of a dynamic link package. The *DosGetModName* call takes the handle of a module and returns the name of its file as a character string. Remember that the filename of that file does not have to be the same as the module name retained by the Loader and input to *DosLoadModule* or *DosGetModHandle*.

Remember also that, so long as a package (or program) is loaded in storage, its executable file is held open by the Loader. It is not possible to erase or rename one of these files, or to open it for output.

4.5.4 Getting Exported Addresses

Although *DosLoadModule* binds a package to the current process, it does not give any specific addresses. The *DosGetProcAdr* call returns a far pointer to any one of the entry points exported by the package (any of the points named in the *EXPORTS* list in its Linker definition file).

This system call takes a module handle and a parameter that specifies which entry point is wanted. That may be a character string, the public name of the entry point, just as it would have been specified to the Linker. (The name must be capitalized exactly right, too; this match is sensitive to letter case.)

It is also possible to ask for entry points by number, or ordinal as it is termed. It is always best to specify an entry point by name when that is possible. The reason is that the ordinal position of an entry point might change from version to

version of a package. The programmer who links the package has to take some trouble to retain the same ordinals from version to version. It is easier for all concerned to keep the names constant and let the ordinal positions fall where they may.

It is not possible to ask for entry points to system packages like VIOCALLS by entry name; the names are not retained in the loaded module (a silly kind of storage economy, not at all justified in my opinion). If you know the correspondence between a system function name like *VioWrtCellStr* and the ordinal entry in VIOCALLS, you can get the function's address from *DosGetProcAdr*.

It is not possible to get a procedure address for a module that has not been bound to the current process. One way to find out whether a loaded module has also been bound is to get its handle with *DosGetModHandle* and then use that handle in a call to *DosGetProcAdr* (using an entry name string of *#1* since any package must have an entry with ordinal 1). If the call works, the package is bound to the calling process; if it fails then the package has been loaded (since *DosGetModHandle* succeeded) but has not been bound. A call to *DosLoadModule* will always bind a module.

Reference: exported names in definition file 3.3.4.7

4.5.5 Calling Remote Functions

The result of *DosGetProcAdr* is a far pointer, a doubleword that contains the selector and offset of the exported entry point. It's not easy to declare and call a far procedure indirectly, through its address. In assembly language, it might be done with

```
call  dword ptr ProcAdr
```

where *ProcAdr* is the doubleword variable filled in by *DosGetProcAdr*. In C, the attributes of the remote entry point are declared as part of the declaration of the pointer to it. Suppose it takes a long argument and returns an integer. Then the far pointer variable might be declared as:

```
int (far *)ProcAdr(long)
```

or, if it wasn't a C function and used the Pascal stack discipline,

```
int (pascal far *)ProcAdr(long)
```

The address would be filled in with a call like

```
DosGetProcAdr(modhdl,"ENTNAME",&ProcAdr);
```

and the resulting function would be called using indirection:

```
anint = *ProcAdr(63L);
```

It is more difficult to do these things in Pascal since that language has no standard, portable way to declare and use procedure pointers. Microsoft Pascal recognizes the nonstandard data type *ADSPROC* as being a far pointer to a procedure, and allows the *ADS* operator to be applied to a procedure name so as to capture its address. However, a procedure pointer may only be passed around as a data object; there is no syntax for declaring a parameter list for such a procedure, nor any way of calling a procedure indirectly, via an *ADSPROC* variable.

A Pascal solution can be worked out, but it requires assembly language hacks and results in fairly unsafe code. The assembly helpers are shown in their entirety in Figure 4.1. The entry points named *PUSH...* may be declared as external procedures. A series of calls on these procedures can be used to build up any sequence of parameters on the stack. All they do is return without cleaning up the stack! Clearly they cannot be used safely unless you are very sure the formats of data on the stack, and very certain of what the called procedure will expect. More procedure names may be added after the existing ones if it is necessary to push other types of data on the stack.

The procedure named *FARCALL* takes one parameter, the far pointer of a procedure to be called. It swaps the destination pointer with its own return address, then returns. That sends control to the specified destination; when that remote procedure returns, it will return to the code that invoked *FARCALL*. We must all sincerely hope that the remote procedure will have popped exactly the parameters that were pushed before *FARCALL* was called.

Figure 4.1
Code to Allow Pascal to Make Indirect Calls

```
_TEXT   segment BYTE PUBLIC 'CODE' ; join other runtime stuff
        assume  CS:_TEXT, DS:NOTHING
;
; The following entry points allow parameters to be pushed onto the
; stack from Pascal code.  The gimmick is simple: return without
; clearing the stack!  Other procedures can be defined as needed,
; though these five cover most needs.
;       PROCEDURE PUSHINT(OneWord:integer); EXTERNAL;
;       PROCEDURE PUSHWORD(OneWord:WORD); EXTERNAL;
;       PROCEDURE PUSHREAL(OneDword:REAL); EXTERNAL;
;       PROCEDURE PUSHLONG(OneDword:INTEGER4); EXTERNAL;
;       PROCEDURE PUSHADS(OneDword:ADSMEM); EXTERNAL;
nameless proc far
        public PUSHINT, PUSHWORD, PUSHREAL, PUSHLONG, PUSHADS
PUSHINT:
PUSHWORD:
PUSHREAL:
PUSHLONG:
PUSHADS:
        ret
nameless endp
;
; The following entry point implements an indirect procedure call.
;       PROCEDURE FARCALL(ToWhat:ADSPROC); EXTERNAL;
; On arrival the stack has:
;               (assorted parms left by PUSHxxxx calls)
;                       dword far pointer 'ToWhat'
;                       dword return address
; and all we have to do is reverse the order of the doublewords
; and then return.  We "return" to the target procedure, it returns
; to the code that called us (hopefully popping those parms).
        public  FARCALL
FARCALL proc    far
        push    bp
        mov     bp, sp
        mov     ax, word ptr [bp + 6]
        xchg    ax, word ptr [bp + 2]
        mov     word ptr [bp + 6], ax
        mov     ax, word ptr [bp + 8]
        xchg    ax, word ptr [bp + 4]
        mov     word ptr [bp + 8], ax
        pop     bp
        ret
FARCALL endp
_TEXT   ends
        end
```

5

The Execution Environment

This chapter is about how a program gets information about the system and how it handles errors and unexpected events.

In this chapter we are considering the operations of a single process; in the next major topic we take up the subjects of how one process creates others and communicates with them.

5.1 The State of the System

There are many things that the program can learn about the system. It can find out about the system hardware configuration. It can get information on the date and time in several formats. It can interrogate the identities of threads, processes, and screen groups; it can inspect general information segments; and it can examine the segment of environment strings it inherited from its parent process.

5.1.1 System Configuration

Three system calls yield information about the system and its hardware configuration: *DosGetMachineMode*, *DosGetVersion*, and *DosDevConfig*.

5.1.1.1 Checking the Machine Mode

Many programs, especially ones written in a high-level language, don't care whether they are running in Protect mode or in Real mode. Many others require the full services of OS/2 and so cannot run in Real mode at all.

A few programs support both Protect mode and Real mode, and operate differently when in Real mode. These programs need to know which mode the CPU is in, Real or Protect. An assembly-language program can find out in three instructions:

```
smsw ax ; machine status into AX
test ax,1 ; test mode-bit
jz realmode ; jump if in Real mode
```

but a program in another language may ask OS/2 with the *DosGetMachineMode* call. That call sets a byte to zero in Real mode and to 1 in Protect mode.

References: real mode 1.3.1; protect mode 1.3.2

5.1.1.2 Getting the OS/2 Version

The OS/2 version number is returned by the *DosGetVersion* call. It can be regarded as a binary word or as two binary bytes. The more significant byte contains a major version number. This is initially 10 decimal, so the word value is always at least 2,560 decimal. The major version number should remain constant for some time, changing only with major functional enhancements to the system.

The least significant byte of the word contains a minor version number. It isn't clear how this number will be used. It might change to reflect software maintenance that does not affect the program interface to the system. Or it might be changed by hardware makers to reflect their proprietary extensions to the basic system.

An additional revision letter is available in the global information segment returned by *DosGetInfoSeg*. Its relation to the version numbers isn't clear.

For a program that runs in the first version of OS/2 there isn't much point in checking the version number. Surely any subsequent version will support the programs that run in the original version! The version number will only become important with the advent of a second version that contains new function, or with the arrival of an OEM's proprietary enhancements reflected in the minor number. Programs that require the added functions will have to check the version number to make sure they are *not* running in the original system.

The Microsoft C and Pascal compiler start-up routines invoke *DosGetVersion* and store its output word at public label *__dosverminor* (two leading underscore characters). The more significant byte is at public label *__dosvermajor*.

Reference: program entry code 4.3.2

5.1.1.3 Getting the Device Configuration

The *DosDevConfig* call can answer several different queries about the hardware attached to the system, as noted in Table 5.1. The system finds the number of printer adaptors and serial ports by feeling out the hardware during bootstrap, and it can be fooled by nonstandard device interfaces.

Table 5.1
DosDevConfig Queries and Replies

Query	Returned value
0	Number of printer attachments
1	Number of serial port attachments
2	Number of diskette drives
3	Math Coprocessor? Zero if none, else 1
4	BIOS submodel type byte (01h typical)
5	BIOS model type byte (0fch typical)
6	Color display? Zero if monochrome, else 1

The number of diskette drives is the number known to the BIOS. Information on the types of the drives (that is, which are normal 5-inch, versus high-density 5-inch versus 740KB 3.5-inch, versus 1.4MB 3.5-inch) is not directly available; some of it can be inferred from the *DosQFSInfo* and *DosQCurDisk* calls.

The model and sub-model bytes returned by queries 5 and 4 are the byte values returned by the BIOS of an IBM PC/AT or compatible in response to Int 15h

with AH=0c0h. The model byte will be 0fch in a PC/AT or compatible or an IBM PS/2 Model 50; this covers a wide range of systems including some with Intel 80386 CPUs, so it really says very little about the system capacity.

The query about display adapters distinguishes only between a monochrome display and color; it says nothing about the type of color display and is no help in distinguishing among the wide array of color displays available. The *VioGetMode* call tells more.

References: *DosQFSInfo* 8.5.4; *DosQCurDisk* 8.5.2; *VioGetMode* 10.4.1.2

5.1.2 Clock and Calendar Information

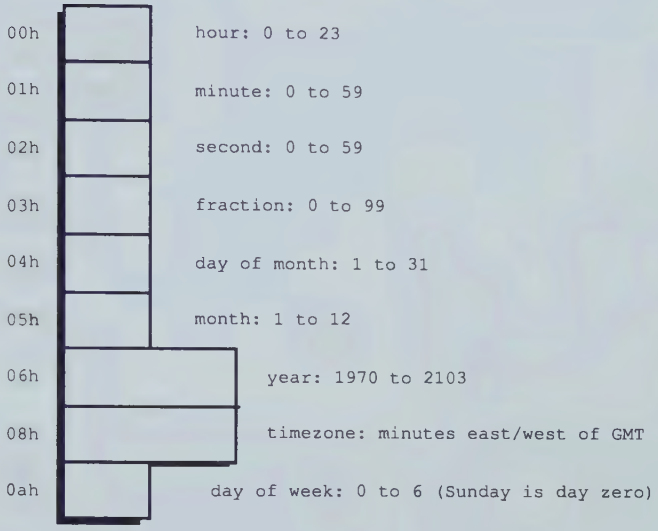
Complete information about the date and time is available from the *DosGetDateTime* call and from the global information segment discussed below. The date and time information is returned in a data structure with the format shown in Figure 5.1.

The date and/or time may be set by passing a structure of the same format to the *DosSetDateTime* call. As with other complementary pairs of system calls in OS/2, the recommended way of setting data is to query it with the Get call, change only the fields of interest, and then pass the updated structure to the Set call. This approach works particularly well with time information, for instance when the hour needs to be advanced for daylight savings time or when an incremental change is to be made in the seconds value to trim the clock to an external standard. A program to set the time by this means is included in the Disk Appendix to this book.

5.1.2.1 The GMT Offset

One field in Figure 5.1 may be unfamiliar unless you have experience in astronomy or radio communications: the time-zone value specified as "minutes east or west of GMT." There is a global standard for time. The current time at the zeroth meridian of longitude—also known as the Greenwich Meridian because it passes through Greenwich, England—is its basis.

Figure 5.1
Format of Date-Time Structure



Greenwich Mean Time (GMT) is the standard time for any use where the scope of interest is wider than the local time zone. The time of an astronomical event, or of an international radio broadcast, or any other moment that must be coordinated across national boundaries, is properly noted in GMT.[1]

It is appropriate that the hour, minute and second maintained by OS/2 should be the local time that is used by people in their daily work. But for some purposes it is also essential to know GMT with precision. It isn't enough to know only the local "time zone." A time zone is a coarse measure, at least 15 degrees of longitude wide—one hour, as the Earth revolves—and the relation between local time and GMT is accurate only at the exact center of the zone. Worse, many time zones have irregular boundaries that don't follow meridians of longitude.

---

[1] *When GMT is referenced to an atomic clock it is called Coordinated Universal Time (abbreviated UCT). The standard for UCT is supervised by the Bureau International de l'Heure in Paris, but UCT is maintained and distributed within each nation by a national agency. In the United States, UCT is maintained by the National Bureau of Standards (NBS) and broadcast by the NBS on radio station WWV throughout the United States. For practical purposes, "GMT" and "UCT" are synonymous.*

The GMT offset carried by OS/2 allows a program to correct the system's time to GMT with good precision. The system's longitude can be specified within a fourth of a degree east or west of the magical meridian. To set it precisely, consult a geographical atlas or a good map to discover the longitude of your location. It is a number of degrees between zero and 180. Multiply by four to determine the number of minutes that separate you from GMT. This is the absolute value that should be stored in the OS/2 timezone word, offset 8 in Figure 5.1.

The sign of the timezone word should be such that when it is added to local time, GMT results. It should be a positive number if your location is west of Greenwich—in the Americas or the Pacific, to the right (as seen on a globe) of the International Date Line. It should be a negative number if your location is east of Greenwich—in Europe, Asia, or the Pacific rim, to the left (on the globe) of the Date Line.

If you have the same trouble I do in remembering how the hour changes around the world, a mnemonic might help. West of Greenwich, GMT is always a later hour than local time. I, a resident of North America, remind myself of this by saying "the old world is older." That reminds me that local time must become later—"older"—to match GMT, so the timezone word should be positive. In the Eastern meridians GMT is earlier than local time, and the timezone word must be negative so as to make local time earlier or younger or smaller. You might key this notion to a mnemonic based on some good-natured slander like "the English are still in bed."

When adjusting the system clock for Daylight Savings time ("summer time" in the U.K.) remember to adjust the timezone word as well. Change it by 60 minutes one way or the other, depending on whether the adjustment will make local time more or less like GMT. For instance, in the United States the change from winter time to Daylight Savings time means putting the clock ahead by an hour. That moves local time an hour nearer GMT, so the timezone word must be reduced by 60 minutes.

5.1.3 Getting Process and Thread IDs

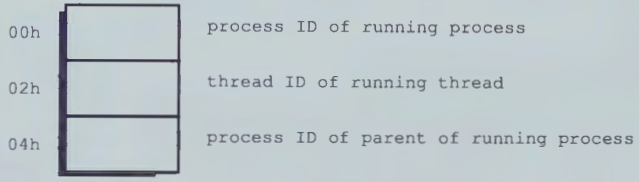
Every process has a binary number, its Process ID or PID. Process ID numbers appear to be assigned consecutively from 1, beginning at bootstrap time, so a process created later will have a larger PID.

Each thread within a process has an ID as well. The first thread in any process has Thread ID (TID) 1, and the TIDs of threads created later are consecutive.

The PID of the running process and the TID of the running thread are available in two ways. The *DosGetPID* call returns them in the data structure shown in Figure 5.2. They are also available in the local information segment returned by *DosGetInfoSeg*, discussed later.

The PID is a good basis for a unique filename for a temporary work file. Convert it to digits and append three letters unique to the program. No other instance of the program can construct the same filename, at least until the system has been booted again.

Figure 5.2
Form of the Structure Returned by DosGetPID



The Process ID of the parent of the running process is returned by *DosGetPID* (and in the information segment). If the program was started from the menu of the Program Selector, this will be the PID of the Program Selector. If the program was started by a command shell, this will generally be the PID of the shell program—although strangely enough, if the program was started with a detach command the parent PID is zero. A detached process is an orphan, apparently.

5.1.4 The Information Segments

The system maintains a single, global information segment and updates it continuously. There is also a local information segment that is maintained for each process. These segments have a wealth of information and deserve careful study.

5.1.4.1 Using DosGetInfoSeg

Access to the global and local information segments is gained through the *DosGetInfoSeg* call. This procedure returns two selector numbers. The first is a se-

lector into the GDT that gives access to the global information segment as a read-only data segment. The second is a selector into the LDT for the local information segment as read-only data.

Naked selectors are not usable as addresses. In assembly language, they have to be loaded into a segment register. In C, a selector has to be insinuated into the most significant word of a far pointer and the least significant word set to zero (the include files in the Disk Appendix define a union, *ADS*, that makes this easier to do). In Pascal, the selector must be assigned to the *.S* component of a variable of type *ADS* and the *.R* component set to zero; then the variable may be used as a pointer.

References: GDT and LDT 2.4.4; selectors and offsets 2.2.4

5.1.4.2 The Global Information Segment

Any process may get a selector for the segment. It selects a descriptor in the GDT that accesses the information segment as read-only data. The content of the segment is diagrammed in Figure 5.3. (There are declarations of it as a data structure in the *SYSINFO* include files for Pascal and C in the Disk Appendix.)

5.1.4.2.1 The Date and Time Fields

The fields related to date, time, and timezone are discussed above in relation to the *DosGetDateTime* system call.

5.1.4.2.2 The System Version Fields

The fields related to OS/2 version numbers and revisions have been discussed above in relation to the *DosGetVersion* call.

5.1.4.2.3 The Huge Shift Value

The Huge Shift used in calculating addresses within huge storage segments is available in the global information segment.

Reference: use of huge shift 7.3.3.1

5.1.4.2.4 The Timer Resolution

Fourteen bytes into the global information segment is a word that gives the resolution of the system timer. The word is an integer in units of ten-thousandths of a second, that is, tenths of a millisecond. In the first release of OS/2 it reads 310 decimal, indicating a timer resolution of 31 milliseconds.

This value marks the limit of precision of the system timestamps and of all other system time intervals: the lengths of dispatching timeslices, the intervals of a timer interval set by *DosTimerAsync* or *DosTimerStart*, and the length of time specified to *DosSleep* or one of the semaphore calls. All of these units must be regarded as plus or minus the resolution interval of the timer.

References: timer services 9.5; semaphore operations 9.4.1

5.1.4.2.5 The Timestamps

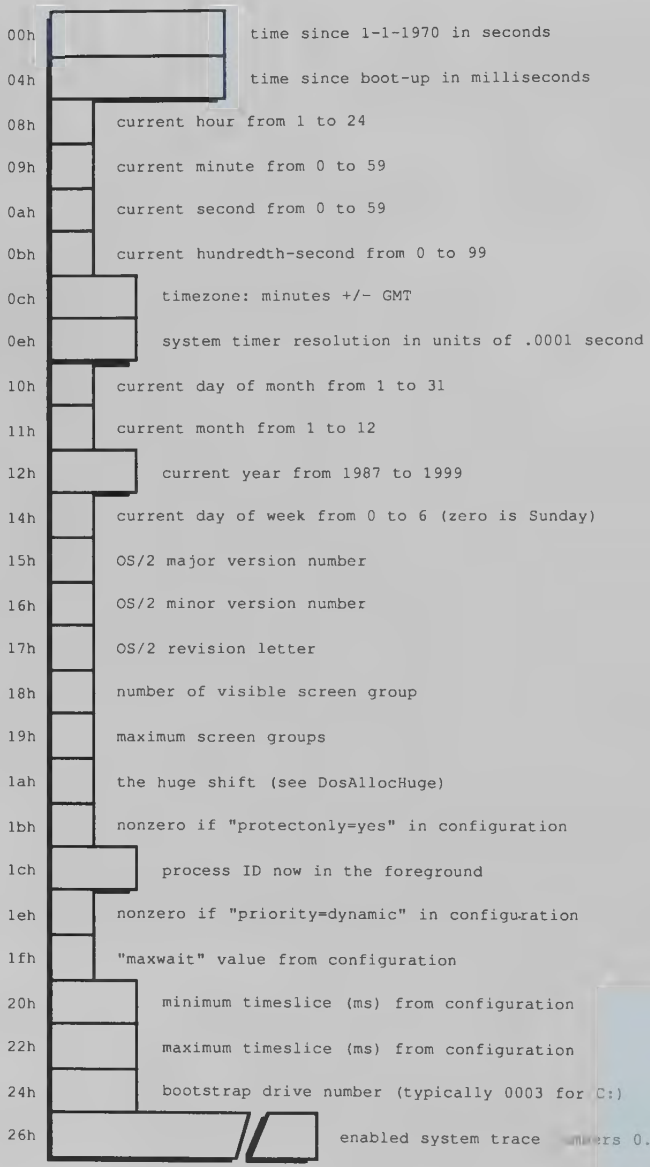
The first two items in the global information segment are timestamps that are updated continuously. They have different precisions and different uses.

5.1.4.2.5.1 The Long-Term Timestamp

The first timestamp is a 32-bit integer that counts the seconds elapsed since 00:00 hours on January 1, 1970. There are 31,557,600 seconds in an average year (60 * 60 * 24 * 365.25) but the precision of a signed 32-bit integer is 2,147 million (2 to the 31st power). Thus it will take 68 years for this timestamp to overflow and become a negative number (the year 2038), and 68 more before it wraps around to zero again.

This long-term timestamp is updated once per second, so it cannot be used to time very short intervals. Its most important use is as a timestamp to distinguish between earlier and later versions of a file or a record. Over the span of years 1970 to 2038, successive values of the timestamp have a proper arithmetic relationship in that "later" is also numerically "larger" in any language that supports signed, 32-bit integers. When unsigned 32-bit integers are supported, as in C, the relation runs to the year 2106.

Figure 5.3
Form of the Global Information Segment



5.1.4.2.5.2 The Short-Term Timestamp

The second field of the global information segment is a 32-bit integer that counts the milliseconds that have elapsed since the system was booted up. Although this timer counts in units of milliseconds, it is only accurate to the timer resolution given in the information segment.

The most significant bit of the short-term timestamp will be set to 1 less than 25 days after the system is booted. The timestamp will wrap around to zero when the system has been up for 49 days of continuous operation. Most OS/2 systems are rebooted sooner than this, but a few that operate unattended might stay up long enough for the timestamp to go negative or wrap. Since the chance exists, a program that treats short-term timer values as integers must take precautions. The "later" of two values might not be "larger"; if the arithmetic signs of two values differ, a correction must be made.

One way of making an anticipatory correction is to perform an unsigned right shift on all values of the short-term timestamp. Since the timer is updated at intervals greater than 16 milliseconds, its least-significant four bits can be discarded without losing much information. The pseudo-code procedure sketched in Figure 5.4 shows how this fact can be used to extend the positive period of the short-term timestamp to more than a year.

Function Stamp must be called whenever a timestamp value is needed. For the first 49 days after bootstrap it returns a copy of the short-term timestamp right-shifted by four bits. The right shift ensures that the number is a positive integer. No information is lost in the shift; if function Stamp returns successive values that are identical, then the original timestamps would also have been identical.

Around the 49th day after bootstrap the real short-term timestamp rolls over to zero and begins counting up again. The next time function Stamp is called it recognizes that this has happened; the fact that the current timestamp, shifted, is less than the sample saved on the last call alerts it to the overflow. The function begins to add a correction factor of 2 to the 28th to all its responses. This preserves the arithmetic relation of "later" and "larger." (Actually, if 49 or more days elapse between calls on function Stamp, it will fail to notice the rollover.)

Figure 5.4
Pseudo-code of Time Function

```
Factor and Sample are static long integers initially zero.
Fixup is a static long integer initially 10,00,00,00h

Function Stamp returns a long integer:
Work is a temporary long integer.

begin
        Work := the short-term timestamp
        shift Work right 4 bits
        if (Work < Sample) /* 49-day wraparound! */
                Factor := Factor + Fixup
        endif
        Sample := Work
        return (Work + Fixup)
end.
```

This function extends the positive period of the short-term timestamp by eight times, to more than a year, without significant loss of precision.

The next six times that the short-term timestamp rolls over, function Stamp increases its correction factor. Not until the eighth overflow, more than a year after the system was booted, will the correction factor overflow into the high-order bit of the result and produce a negative timestamp. If the timestamp is treated as an unsigned 32-bit integer, it will continue to increase for more than two years of continuous operation without a bootstrap, a mark that very few personal computers will ever attain.

5.1.4.2.6 The Configuration Values

Several items in the global information segment reflect values that were specified in the configuration file when the system started up. These include some scheduling parameters for which some programs might have requirements.

5.1.4.2.6.1 Protect Mode Only Status

The byte at offset 27 (01bh) is nonzero only if the statement *protectonly=yes* was in the configuration file. Some programs might require this (perhaps because a Real-mode program can breach security and examine a Protect-mode program's data), and they can check for it in the information segment.

5.1.4.2.6.2 Scheduling Parameters

The scheduling parameters specified (or defaulted) in the configuration file appear in the information segment. The byte at offset 30 (01eh) is nonzero only if *priority=dynamic* was allowed. Dynamic priority means that a thread that has been waiting to run for the maximum wait time will have its priority boosted. The maximum wait time in seconds (from the configuration file *maxwait=* statement) can be found at offset 31 (01fh).

The minimum (and normal) timeslice interval appears at offset 32 (020h) as an integer in units of milliseconds. If the *timeslice* configuration statement was defaulted, this will probably be the same as the timer resolution interval. The maximum timeslice interval follows the minimum, also as a word of milliseconds. The use of these parameters is discussed in the chapter on asynchronous processing.

Reference: timeslicing 9.1.6

5.1.4.2.7 The Foreground Screen Group

At offset 24 (18h) is a byte that holds the number of the screen group that is in the foreground. The foreground screen group is the one that is visible on the display and that is connected to the real keyboard and mouse. This is not necessarily the same as the screen group used by the running program! The running program's screen group number appears in the local information segment discussed below.

At offset 25 (19h) is a byte giving the maximum number of screen groups. It is probably 16 (10h). The significance of this value is that screen groups are numbered starting from zero, so no screen group will have this number, and this number is used elsewhere to mean "no screen group."

Reference: screen group 1.2.1

5.1.4.2.8 The Last Keyboard User

At offset 28 (1ch) is a word that is referred to as the ID of the foreground *process*. This is confusing; it is not at all clear what it would mean for a process to be in the foreground; the foreground *screen group* is the one that owns the physical hardware, but a screen group may be occupied by many processes.

It turns out that the word at offset 28 contains the PID of the process that most recently read from the physical keyboard. That process might or might not be associated with the foreground screen group.

References: process ID 5.1.3; keyboard input ch. 11

5.1.4.2.9 The Trace Flags

OS/2 contains an internal trace facility that is intended for use only by a select circle of maintenance programmers and diagnosticians, and therefore is not documented for general use. This much is known: the OS/2 kernel maintains a storage buffer in which it records a circular trace of records. Each record contains a major and a minor trace-code number and some trace-dependent information.

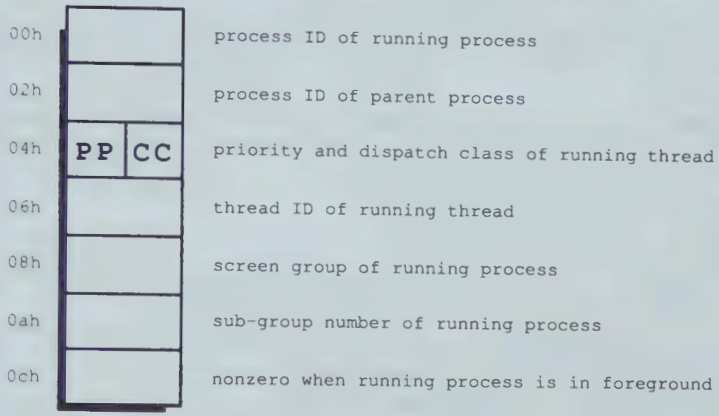
A variety of events and actions in the kernel can be traced, and these traces may be selectively activated. The location of the trace buffer isn't known, nor is the system interface that gains access to it (though there must be one). It wouldn't be much use to locate the buffer, since the format of the trace records is not documented.

There are potentially 256 different major trace codes, although how many are implemented is not known. The final 32 bytes of the global information segment, beginning at offset 38 (26h), contain a set of 256 bits. Each bit represents one trace type. If only major trace code zero is enabled, the byte at offset 26h contains 80h; if only major code 1, 40h, and so forth through major code 255.

5.1.4.3 The Local Information Segment

The local information segment has some useful numbers that apply to the running process and thread. The layout of the segment is shown in Figure 5.5.

Figure 5.5
Format of the Local Information Segment



5.1.4.3.1 Process and Thread IDs

The Process ID of the running process, its parent process, and the thread ID of the running thread, are in the local information segment. These items are also available by calling *DosGetPID*.

The last word of the local information segment is zero unless the current process is the one that last read from the physical keyboard. Then it contains -1 (all-ones binary).

Reference: *DosGetPID* 5.1.3

5.1.4.3.2 Screen Group Number

The number of the screen group of which the running process is part, appears in the local information segment. If that number is the same as the screen group listed at offset 018h in the global information segment, the screen output of the running process is visible to the operator—and if it is not the same, it is not visible.

If the screen group number is equal to the number given in the global information segment at offset 019h, then the running process *has no screen group*. The process may have been started with a *run* statement in the configuration file, or

with a Detach command from the command shell. In either case, it has no access to a display, a keyboard, or a mouse. Standard output will be going either to a disk file or to the null file, and standard input will either be null or from a disk file. (A detached process may display urgent messages to the operator in a pop-up display, as discussed in the chapter on display output.)

The local information segment also contains a field specified to be a subscreen group. It is hard to discover what this means; it has been zero in all my trials.

Reference: pop-up displays 10.4.4

5.1.5 The Environment Segment

Every process has an environment segment. When the process is first dispatched, the selector for the segment is in register *AX*, but it can be recovered at any time with a call to *DosGetEnv*.

The environment segment contains two blocks of character strings. The strings in the first group have a conventional format and many important uses as system parameters. The strings in the second group convey program parameters, usually from a command line.

5.1.5.1 String Format

The two blocks of data in the environment have the identical format. Each consists of zero or more ASCIIZ character strings. An ASCIIZ string is a sequence of any number of nonzero bytes (conventionally these are printable characters, but they don't have to be) followed by a byte of zero to mark the end.

A byte of zero by itself is used to mark the end of a block of strings. A lone byte of zero is a null ASCIIZ string (one with no contents), so it could be said that a null string marks the end of a block of strings. Also, it is not possible to have a null string within the environment segment; it would be taken as the mark at the end of the block. The nearest approach to a null string would a string containing a single blank.

The smallest block of strings is a single byte of zero. The upper limit on the total size of the environment segment is 64K bytes.

5.1.5.2 Environment Variables

The first group of strings in the environment segment constitutes a pool of named variables. Each string in the pool should have the form *name=value*.

By convention, the *name* should be spelled in uppercase letters. The *value* may be any ASCII characters (in fact, any byte values at all) except for the null character, 00h. The null character terminates the string.

Any regular user of OS/2 or MS-DOS is familiar with the variables that are conventionally found in the environment:

- *PATH=* a list of path-prefixes to be searched for executable files
- *DPATH=* a list of path-prefixes to be searched for data files
- *LIB=* the path of the directory where the Linker will search for object libraries
- *INCLUDE=* the path of the directory where the assembler and most compilers will search for included source files

Many programs take specific parameter lists in the environment; the assembler for example takes default options in the *MASM=* variable.

5.1.5.2.1 Locating Named Variables

At the risk of being too obvious, I offer in Figure 5.6 the outline of how all environment variables can be scanned in sequence. The key to the main loop is that, if the first byte of the next string is the null byte, the process is finished. This must be a while-loop because the environment can be empty.

OS/2 contains a system procedure that simplifies processing the environment. The *DosScanEnv* call takes a name string, searches for it, and returns a far pointer to the value string that follows it. It provides the simplest way of finding the value of a variable.

Figure 5.6
Pseudo-code of an Environment Scan

```
Name and Value are far-pointers to character

Set Name to the address of the Environment segment

While (*Name <> 00h)
        Value := Name
        Repeat
                advance Value one byte
        Until (*Value == '=')
        advance Value one byte
        Process (Name->name,Value->value)
        Name := Value
        while (*Name <> Null)
                advance Name one byte
        end while
        advance Name one byte (start of next)
end while
```

The *DosScanEnv* call returns an error code if it fails to match the given variable name. However, in this case it does not update the contents of the far pointer variable it was given. The easiest way to use it is to set that pointer initially to the address of a default value-string. Then the return code of *DosScanEnv* may be disregarded; the pointer variable points either to a value or to a default.

5.1.5.2.2 Searching Lists of Paths

OS/2 contains another function to help in dealing with environment variables like PATH, whose values are lists of paths. The *DosSearchPath* function takes the name of a file and a list of paths and locates the first matching file along the list.

Reference: *DosSearchPath* 8.6.5.1

5.1.5.3 Environment Parameters

The second block of strings in the environment usually follows immediately on the first group. Its offset within the environment segment is passed to a new process when it is first entered, and can be recovered later with *DosGetEnv*.

This block is a set of whatever strings the parent of the current process chose to pass as parameters. There may be none, or one, or more than one string. There is no particular format for them.

When the OS/2 Protect-mode command shell, *CMD.EXE*, invokes a program it passes two parameter strings. The first is the command verb, a string that contains the first unit of text from the command line, just as the operator typed and punctuated it, including upper- and lowercase letters. This is normally just the program name, but if the operator typed a complete pathname for the program, the whole path is there. The entry code of a C program copies this string to the heap and puts the address of the copy in the first entry of the *argv* array.

The second string that *CMD.EXE* passes consists of whatever followed the command verb on the command line, exactly as typed by the operator. If nothing at all followed it, the second string will be null. If blanks or tabs followed it, the second string will contain these. The entry code of a C program parses this string into tokens, copies the tokens to the heap, and puts address of each copy into the *argv* list.

The Disk Appendix to this book contains a Pascal include file that locates the argument strings and parses them into an *ARGV* array in the style of C.

Reference: registers at program entry 4.3.2

5.1.5.4 Updating the Environment

The environment segment is read-write data and it may be modified in any way the process chooses. It is ordinarily only large enough to hold the strings that were passed in it; before adding new variables the program will have to expand the segment with *DosReallocSeg*. Also the value of the offset to the parameter strings is a constant kept by OS/2 for the process. It will continue to be returned by *DosGetEnv*, even after the contents of the environment segment have been modified so as to invalidate it.

If the aim of updating the environment is to insert or change variables and then find them with *DosScanEnv* or *DosSearchPath*, the environment segment will have to be altered in place, since these functions use the environment segment selector that is known to OS/2. In this case the only method is to enlarge the segment and then update its contents directly.

When the aim is to prepare an updated version of the environment to be passed to a child process, it is probably best to copy the environment to a new location and then update the copy. That is what the C runtime function *putenv()* does, for instance.

References: *DosReallocSeg* 7.3.6; the environment of a child process 6.1.3

5.2 National Language Support

OS/2 contains a hodge-podge of features that allow a program to tailor its operations to suit local customs. Some of these are inherited from MS-DOS with little change; others are new.

5.2.1 Country Information

The *DosGetCtryInfo* call returns the data structure shown in Figure 5.7. The structure is very similar to one maintained under MS-DOS. The fields in it enable a program to adjust the form of a printed report to agree with local customs in punctuation.

The data in this structure are taken from the file *COUNTRY.SYS*, which is distributed with OS/2. That file has similar information for each of the countries listed in Table 5.2. The data for one particular country are selected with the *country* statement in the configuration file. The system, as part of initialization, indexes the *COUNTRY.SYS* file by the country number and loads the national data that is will return to a *DosGetCtryInfo* call.

The Country Information structure contains answers to many small, vexing questions. For example, the third field (at offset 04h) and the seventh field (offset 0fh) together answer the question, "What's the proper format for the date?" The three elements of the date should be displayed in the order indicated by the third field, and they should be punctuated with the character in the seventh field.

Figure 5.7
Format of the Country Information

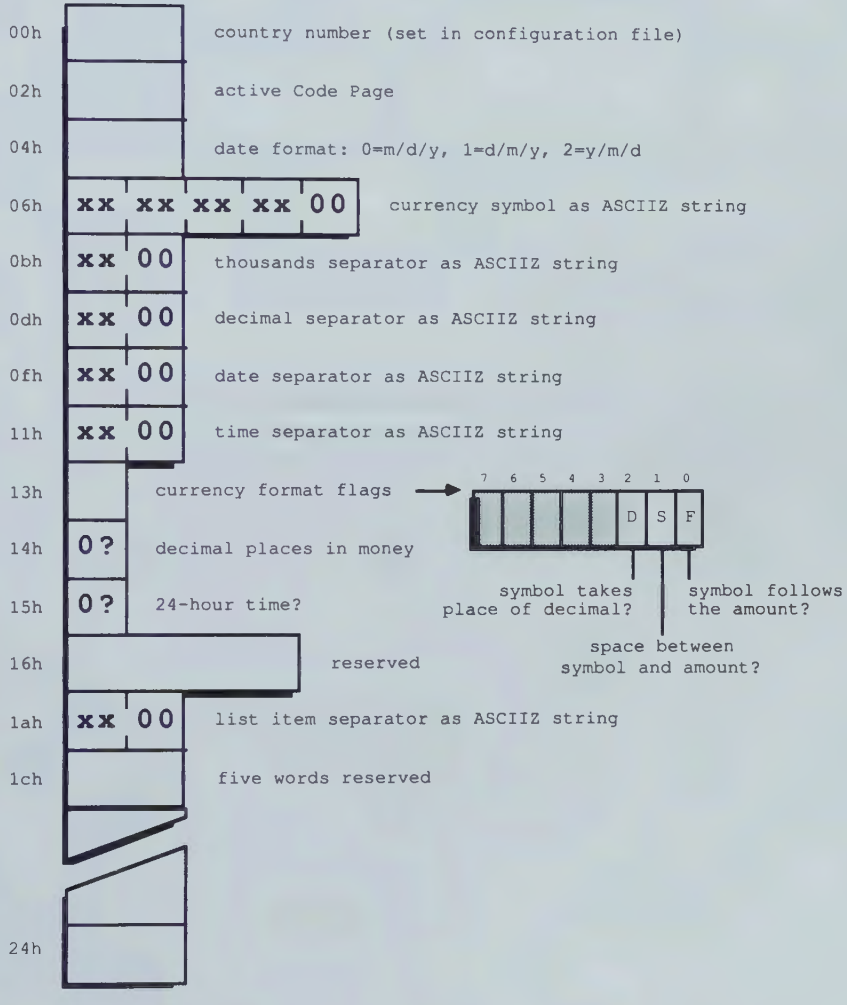


Table 5.2
Country Numbers Recognized by OS/2

Country Number	Linguistic/Geographical Area
001	United States
002	Canada (French)
003	Latin America
031	Netherlands
032	Belgium
033	France
034	Spain
039	Italy
041	Switzerland (both French and German)
044	United Kingdom
045	Denmark
046	Sweden
047	Norway
049	Germany
061	Australia
081	Japan
082	Korea
086	People's Republic of China
088	Taiwan
351	Portugal
358	Finland
785	Arabic language
972	Hebrew language

The proper format for money amounts is always a problem; the Country Information answers it. The currency symbol, which may have as many as four bytes, is given. The proper punctuation for thousands and for the decimal is shown (some Americans are blissfully unaware that in many parts of the world, the amount "eight thousand twenty-six and 42 one-hundredths" would be written *8 026,42* instead of the *8,026.42* with which they grew up). The bit flags at offset 13h document several different kinds of money formats. If bit 2 is set, the

currency symbol given is to be written in place of a decimal separator in money amounts (what countries do that, do you know?). If it isn't set, then bit zero specifies whether the currency symbol precedes or follows the amount, and bit 1 tells whether it should be closed up or spaced away from the numerals.

5.2.2 Handling ASCII Differences

There are slight differences in the meaning of ASCII from one country to another. Some of these are addressed by two services, *DosCaseMap* and *DosGetCollate*.

The *DosCaseMap* service converts letters to uppercase (map is a mathematician's term for translate) under the rules of a specified country. It can presumably handle double-byte character sets (see below) as well, which may not be the case with other subroutines for uppercasing.

The *DosGetCollate* service returns a table that can be used for sorting ASCII text. Each byte in the table represents the collating value of the corresponding byte of ASCII. If the collating table is named *TABLE*, and if there are two ASCII byte values *X* and *Y*, *X* should be sorted ahead of *Y* when *TABLE[X]* is less than *TABLE[Y]*. The collating table ignores letter case, so that *a* and *A* have the same collating value. This is convenient; it saves having to convert the data to uppercase before collating it.

5.2.3 Testing and Setting the Code Page

Code page is a very ambiguous term in OS/2 as it was in MS-DOS. In many cases a code page is a translate table that relates ASCII codes to the symbols input or output by a device. The *KbdSetCP* service, for instance, sets the translation between keystrokes and ASCII, while the *VioSetCP* service sets the translation from ASCII to characters on the display screen. These code pages have radically different formats. They might be called *device* code pages, and in fact are stored in files with the extension .DCP.

There is another code page that sets the logical translations to uppercase and collating sequence. The number of this code page is returned by the *DosGetCP* call. A different one can be set using *DosSetCP*.

References: *KbdSetCP* 11.7; *VioSetCP* 10.4.1.5

Table 5.3
Code-Page Numbers Supported

Code Page Number	Language Group Supported
437	American English
850	General European
860	Portuguese
863	Canadian-French
865	Nordic

5.2.4 Double-Byte Character Sets

The character sets of some languages will not fit in the confines of the 128-byte ASCII set. OS/2 contains support for extended character sets in which some characters are expressed as two bytes. These are called Double-Byte Character Sets and abbreviated DBCS. The keyboard input functions and display output functions can be made to handle DBCS characters given the correct device code pages. Once the extended characters are in storage, they must be interpreted; to be interpreted they must be recognized.

The *DosGetDBCSEv* call returns a table that helps to interpret DBCS text. It returns a variable number of pairs of bytes. Each byte specifies a range of byte values. An ASCII byte that falls into one of these ranges is the lead-in byte of a double-byte character.

Ordinarily, *DosGetDBCSEv* will return only a pair of zero bytes, signifying that there are no DBCS characters. When a DBCS code page is active, it will return one or more pairs of nonzero bytes, followed by a pair of zero bytes to indicate the end of the table.

5.3 Unplanned Events

A process may be affected by unplanned events. Some of these events are initiated *outside* the process, for example the pressing of a Control-break keyboard signal. Other unplanned events arise from an action of the program, for example the result of attempting to divide by zero. The program can set up special rou-

tines to handle many of these events. There are two different mechanisms, one for signals from outside and one for exceptions from within.

Some events cannot be intercepted by the program (for example, the exception that arises from addressing a nonexistent segment), and the program may choose not to intercept others. There is a third, general mechanism by which an Exit routine can be set up to end the program in an orderly way.

5.3.1 Signals From Without

There are six signals that can arise outside the process. The system sets up a default handler for each one of them. However, the program may delay the signals during critical processing, or block them, or set up its own handlers to intercept them. Signal policies may also be nested, so that a library routine, say, can set up a temporary handler for one signal, block others, and later restore the handling of all signals to its previous state.

5.3.1.1 The Outside Signals

The outside signals are:

- Control-C pressed
- Control-break pressed
- Process termination requested
- External signals A, B, and C

The Control-C and Control-Break signals are quite similar. The Control-C signal can be recognized only while the keyboard is being read in cooked mode, but that is the normal way of reading the keyboard. Control-break may be recognized at any time. Many programs need to block termination by these signals, and OS/2 makes it easy to do so.

The signal that somebody wants this process to end arises when the *DosKillProcess* system call is invoked with this process's ID. Why would one process want to terminate another one? It turns out that most often, the *DosKillProcess* service is used to commit suicide, not murder.

The signals *A*, *B*, and *C* are generated when another process issues the *DosFlagProcess* call, naming this process. They are not a threat to a process's existence; they are a communications path. The flag signals enable one process to send an

event to another in an abrupt, peremptory way. They might be used, for instance, by a communications handler to notify a communicating program of a lost connection.

5.3.1.2 Holding Signals

All six signals as a group can be *held,* that is, delayed, while the process performs some critical work that should not be cancelled or interrupted. The *DosHoldSignal* call is used to do this.

Receipt of signals is governed by a count that is maintained for each process. The count is initially zero. So long as it is zero, signals may be received. The *DosHoldSignal* call can be used to increment or decrement the count.

The count mechanism makes it possible to use *DosHoldSignal* within library routines and dynamic link packages. When a package routine is entered to do some critical work, it may issue *DosHoldSignal* to hold signals, that is, to increment the count. When it is finished, it may issue *DosHoldSignal* to permit signals, that is, to decrement the count. However, that only restores the condition in existence when the package was called. It enables signals only if signals were enabled when the routine was entered.

A signal that is held is not rejected or ignored. If a signal arrives it remains pending until finally it can be received. If a second signal of the same type arrives, it is rejected. Only one signal of each type may be pending at one time.

When the signal hold-count is reduced from 1 to zero, any pending signals are taken at once. Thus a call to *DosHoldSignal* might be followed instantly by program termination or by entry to a signal handler.

5.3.1.3 The Default Signal Handlers

Ordinarily an arriving signal is not held but accepted. To accept a signal means to invoke a *handler* for the signal. A handler is a subroutine that receives control when a signal arrives. The handler is entered under the first thread of the process, the one with Thread ID 1. This thread is diverted from whatever it was doing and made to call the handler. When the handler returns, the first thread resumes whatever it was doing.

The system supplies default handlers for each of the six outside signals. Unless the program sets up its own handlers, the default handlers get control whenever a signal is accepted.

The default handlers for the Flag *A*, *B*, and *C* signals simply accept the signal, acknowledge it (acknowledgments are described below), and exit restoring all registers. The first thread never knows it was diverted.

The default signal handlers for Control-C and Control-Break issue *DosKillProcess* for the process. This is how these keyboard signals achieve their deadly effect—the system doesn't kill the process, the default signal handlers do it.

The default signal handler for the process termination signal issues a *DosExit* call with a parameter requesting termination of all threads. This is how *DosKillProcess* actually kills a process: the default signal handler issues *DosExit*.

Pause and consider what happens when the operator types a Control-C. Some system process generates a Control-C software signal destined for the running process. If it hasn't been held or blocked, the signal handler is invoked. If that's the default handler, it issues a *DosKillProcess*. But that action only generates another signal, a process termination signal. If that signal isn't held or blocked, its handler is invoked. If that's the default handler, it uses *DosExit* to end all threads.

Reference: *DosExit* 4.3.3

5.3.1.4 Ignoring Signals

The *DosSetSigHandler* call can be used to cause any of the six signals to be ignored. An ignored signal is in fact redirected to a special handler that accepts it, acknowledges it, and exits. That's the initial state of the Flag *A*, *B*, and *C* signals; they are set to be ignored by default.

The program may use *DosSetSigHandler* to make Control-C, Control-Break, or process termination be ignored. Since the first two don't do anything but cause the third, it suffices to ignore process termination. That makes the process proof against involuntary termination from outside causes.

5.3.1.5 Blocking Signals

The *DosSetSigHandler* call can be used to cause any of the six signals to be rejected. A rejected signal is not accepted, even to be ignored. The process that issues the signal receives an error code, and the target process never feels any effect.

There is little difference between blocking the Control-C, Control-Break, and termination signals, and ignoring them. The processes that issue these signals don't usually check the error code that comes back from the call—although if you write a call to *DosKillProcess*, you can check for the blocked error code afterward and know that the target process was armored against it and hasn't ended.

The main use of blocking a signal is in blocking the Flag signals. A process that attempts to send a Flag to another process will presumably check the return code, and ought to be able to learn something from the fact that the target process was blocking the signal, as opposed to not existing or not responding.

5.3.1.6 Handling Signals

The *DosSetSigHandler* call may be used to establish a handler for any of the six signals. As noted earlier, a handler is simply a procedure that is called automatically when a signal is accepted. The handler receives a one-word parameter on the stack. A handler might be declared in C as:

```
void far pascal SigHdlr(unsigned parm)
   { do something about the signal }
```

and in Pascal as:

```
procedure [PUBLIC] SigHdlr(Parm: word);
   begin
      do something about the signal
   end;
```

More likely, a signal handler will be written in assembly language. That's because the first thread of the process will be diverted to make the call, as if it had executed a call to the handler procedure. But this isn't a normal call; the first thread does not know it is going to make a procedure call at that point. If the procedure modifies any of the machine registers—including the machine flags—the effect on the first thread will be as if lightning had struck its registers. Only

in assembly language can you write a procedure that saves and restores all registers.

There is not much point in setting up a handler for any of the three dangerous signals.[2]

Usually a handler would be set up for the Flag signals *A*, *B*, and *C*. The handlers for these signals receive the parameter that the sending process passed on the *DosFlagProcess* call. They can save the parameter and do something—perhaps clear a semaphore—to let the rest of the process know that the signal has arrived.

5.3.1.7 Acknowledging Signals

From the moment that a handler is invoked, signals of its type are held. If another signal of that type arrives it remains pending. If a third signal arrives it is rejected.

The handler, if it is not going to terminate the program immediately, should use that all-purpose service *DosSetSigHandler* to *acknowledge* the signal. Acknowledging the signal releases the hold on the signal type. If another signal is pending, it will be accepted at once. That may mean that the handler is reentered before it has returned, which may or may not be a problem.

5.3.1.8 Restoring the Default Handler

The *DosSetSigHandler* call may also be used to set default handling for any of the six signals. This amounts to setting up a new handler which happens to be the default handler.

5.3.1.9 Nesting Signal Policies

When *DosSetSigHandler* is called to ignore, or block, or set a handler for a signal (including a default handler), it returns two items: the address of the previous handler and a number indicating the policy previously in force (ignoring, block-

---

[2] *Although there is a certain appeal in the thought of writing a handler for Control-C that uses VioWrtTTY to display a rude message—perhaps "Nyah, Nyah, can't kill me," or "Do that again and you'll be sorry."*

ing, handling). These two values can be used later to reestablish the previous signal-handling policy. If they are passed to *DosSetSigHandler* as input parameters, the result will be a return to the previous state.

This means that a subroutine (especially one in a dynamic link package) can establish its own signal policy while it does some critical work, then restore its caller's signal handler when it exits.

5.3.2 Handling Internal Exceptions

The CPU hardware traps a variety of errors and exceptional conditions. The effect of a hardware trap is to transfer control of the machine to the OS/2 kernel. The kernel's usual response is to terminate the process with a message, for example the message "DOS1811 The process has stopped, the trap number is 13" that follows an attempt to access storage with a bad selector or offset. Termination for a hardware exception is not converted into a termination signal that can be fielded by a signal handler as described above. A process that causes a hardware trap is no longer trusted to manage its own affairs.

Six hardware traps may be handled by a process on its own behalf. If they are not handled, the kernel cancels the process with a message. But if they are handled, the process may be able to take corrective action or end in an orderly way.

5.3.2.1 The Internal Exceptions

These are the hardware exceptions that a process may handle, with their trap numbers:

- (0) Integer division by zero
- (4) Arithmetic overflow as detected by the *INTO* instruction
- (5) Exceeding array bounds as detected by the *BOUND* instruction
- (6) Executing an invalid operation code
- (7) Absence of the Numeric Coprocessor Extension (80287 NPX)
- (16) Floating-point error reported by the NPX

These are exceptions that a sophisticated application—or a well-designed runtime library for a programming language—might reasonably be expected to recover from. We will examine them after studying the conditions under which a handler is entered.

5.3.2.2 Interrupt Handlers

The *DosSetVec* call establishes a handler for one of the hardware exceptions. It takes the address of the handler and returns the address of the previous handler, so that a library routine may restore the previous trap policy when it exits.

The handler for a hardware trap is a procedure that is called automatically when a trap occurs. In this it is like a handler for a software signal, but the two are called in very different conditions and must not be confused. A signal handler is always entered on the first thread of the process. A hardware trap handler is entered on the thread that encountered the exception. One implication of this fact is that, in a multi-threading program, a trap handler could be entered concurrently under different threads. Either the handler must be written to work in this case, or it must begin with a *DosEnterCritSec* call to prevent multiple entrances.

Another difference is that, while a signal handler is running, signals of that type are held. While a trap handler is running, traps of the same or any other type may happen and will cause their handlers to be entered. If the code of a handler causes its own trap, it will be recalled to handle the trap.

A handler is entered by diverting the thread that had the exception. When the handler receives control there are three words on the stack: first the machine flags word, then a far return address. This is the condition under which a hardware interrupt handler is called. The IRET instruction is designed to exit such a routine; it reloads a far return address and also reloads a word of flags.

In some cases the return address points to the first byte of the instruction that caused the exception. That will usually be the operation code byte, but if the instruction had a segment override prefix byte, that is the first byte—and if had a *LOOP* prefix, that is the first byte. When the handler needs to examine and interpret the instruction, it has to allow for these things.

A handler may aim at one of two goals, depending on the kind of trap and on the application. It may aim at correcting the error and restarting the interrupted code in such a way that it will never know about the diversion. Such handlers will generally have to be written in assembly language.

Alternatively, the handler may only expect to diagnose the error in a helpful way, or perhaps to divert the program into an error routine (for instance, when implementing the *ON ERROR* statement of PL/I). One general method of such a routine is to tinker with the stack so that the handler may return to a diagnostic routine instead of the interrupted code. A routine of this sort might be written in C

or Pascal (with an assembly routine to help hack the stack). In that case, the handler would be declared as a far procedure with a single parameter of an unsigned integer (the flag word).

One strategy shouldn't be overlooked. When the handler decides that it cannot recover the error and the process should end, it might do so by issuing a *DosKillProcess* call against its own process ID. That converts a hardware exception into a software signal. The handler can check the error code returned. It might indicate success, or that the signal had been blocked, or that it had been rejected because another termination was already pending. Each of these could suggest different courses of action depending on the design of the application.

References: *DosEnterCritSec* 9.3; machine flags 2.5; *DosKillProcess* 5.3.1.1

5.3.2.3 The Divide-by-Zero Exception

DIV and *IDIV* are the machine instructions that divide one binary integer into another. If the divisor is zero, trap zero occurs. When the handler gets control, the return address points to the first byte of the instruction (or its first prefix byte).

The handler for this exception will usually only issue a diagnostic and terminate the process. In some applications, though, it might set an artificial value for infinity (or zero, depending on the preferred interpretation of division by zero) in the registers, and restart the interrupted code. To do so, the handler has to interpret the instruction (*IDIV* or *DIV*? Byte or word operation?) and then adjust its return address to point to the byte after the instruction.

5.3.2.4 The Overflow-Detected Exception

Several machine instructions for integer arithmetic may set the Overflow flag in the machine flags. In some applications it can be ignored. In other cases it can mean that the calculated result has become nonsense! However, overflow is a rare condition and in the past it has often been ignored on the grounds that it is too expensive to test for it. The designers of the 80286 CPU attempted to reduce the cost of testing for overflow by providing the *INTO* (INTerrupt on Overflow) instruction. It is small (one byte), needs no relocation, and executes quickly (3 clock cycles) in the usual event that the Overflow flag is not set.

When *INTO* is executed with the Overflow flag true, a trap 4 takes place. The return address on the stack points to the byte *after* the *INTO* instruction.

A good compiler will insert an *INTO* after every arithmetic operation on a signed integer, and will provide a handler for interrupt 4 that issues a reasonable diagnostic message. In a sophisticated language like APL, the handler for interrupt 4 might convert the operands of the expression into floating point and continue.

5.3.2.5 The Array-Bounds Exception

The CPU supports a *BOUND* instruction that compares a word in a register to two adjacent words in storage. If the register is less than the first word, or greater than the second, a trap 5 occurs. The return address points to the first byte of the *BOUND* (or its first prefix).

A compiler can use the *BOUND* instruction to make fast, simple tests of array subscripts (13 clock cycles, less than 2 microseconds per check). The program entry routine supplied by a Pascal or Modula compiler might establish a handler for this exception; the handler would issue a diagnostic and end the program.

There are other uses for *BOUND*, of course. A data acquisition program could use it for speedy validation of words of input, with the trap handler inserting a default value for bad ones.

5.3.2.6 The Invalid Opcode Exception

There are only a few byte values that are not valid operation codes to the 80286 (and even fewer to the 80386). However, the CPU will also call some illogical or impossible combinations of opcode and operand invalid. When it rejects an opcode, the CPU raises trap 6. The return address points to the invalid instruction.

In earlier hardware environments, an invalid opcode was usually the result of the program's having taken a wild branch into a block of data. That's less likely in Protected mode, since an attempt to execute data causes a different trap (one that can't be handled). It is still possible to branch into data, but only into constant data embedded in a code segment. Usually a trap handler for trap 6 will only terminate the process.

5.3.2.7 Numeric Processor Exceptions

Two very different kinds of exceptions relate to the 80287 (or 80387) Numeric Processor Extension (abbreviated NPX). If this chip exists, it implements floating-point arithmetic functions that are commanded with instructions that begin with a unique opcode value. But not all systems have an NPX installed.

During system initialization the operating system has to find out whether an NPX exists. If it does, OS/2 sets a bit in the Machine Status Word (MSW) register, a hardware register in the CPU. Whenever the CPU encounters an NPX instruction prefix, it consults the MSW. If the NPX-exists bit is true, the CPU processes the instruction. If the bit is false, the CPU takes trap 7 instead. The handler's return address points to the NPX instruction.

When the NPX exists, a program's instruction can set a floating-point calculation going in it. These proceed asynchronously to the normal CPU instructions, and they can take quite a bit of time. If the NPX finishes its operation successfully, it waits for more work. But if it runs into an error, it signals the CPU and the CPU takes a trap 16. In this case, there's no predictable relation between the handler's return address and the instruction that started the calculation that failed.

Clearly, traps 7 and 16 are mutually exclusive. If there is no NPX it can't cause a trap 16, and if there is one trap 7 can't happen. A program that requires an NPX to exist would set a handler for trap 16. In the handler (which would be written in assembly language) it would interrogate the status registers in the NPX, then either restart the failed operation or end the process with a diagnostic.

A handler for trap 7 has quite different objectives. It receives control on every NPX instruction. Presumably it will attempt to emulate the missing NPX operation in software, then restart the interrupted code.

OS/2 imposes a peculiar restriction on handlers for trap 7. Recall the CPU hardware takes trap 7 based on a bit in the MSW. Each process has its own MSW, loaded by OS/2 when it dispatches (a thread of) the process. As soon as a process establishes a handler for trap 7, OS/2 turns off the NPX-present bit in the MSW for that process—even if an NPX exists! If the trap handler is discarded (by reestablishing the original, default handler), the MSW for the process is set to reflect the existence of the NPX.

The result of this odd rule is that a trap-7 handler cannot be used to find out *if* the NPX exists. You cannot, for example, set up a handler for trap 7 and then exe-

cute an *FNOP* instruction, expecting to enter the handler if the NPX is absent and not if it exists. The handler will always be entered.

If the application is such that it *can* use software emulation *if* the NPX is missing, but would rather not, then it must test the MSW *before* establishing the handler. Similarly, there is no point in setting up a handler for trap 16 if the NPX does not exist. In Figure 5.8 is the complete text of an assembly language subroutine that interrogates the MSW and returns "true" if an NPX exists.

Figure 5.8
Assembly Function to Test for NPX

```
; Declare this function in C as
;       extern int far pascal QUERYNPX(void);
; and in Pascal as
;       function QueryNPX : Boolean; external;

        .286p   ; enable "smsw" opcode

_TEXT   Segment  byte public 'CODE'

        public   QUERYNPX

QUERYNPX proc far
        smsw    ax      ; machine status to AX
        shr     ax, 1   ; NPX in hardware to bit zero
        and     ax, 1   ; isolate it
        ret             ; return boolean in AX
QUERYNPX endp
_TEXT   ends
        end
```

5.3.3 The Exit List

Every process has an associated *exit list*, a list of procedures that are to be invoked when the process has terminated. Initially, the list is empty. The *DosExitList* call is used to add or drop a procedure's address from the list. A library package (whether dynamically linked or not) might set up its own exit routine when it begins supporting a client. Later, if the client finishes all work related to that package, the package might remove its exit routine from the list.

When the last active thread of a process finally terminates, OS/2 inspects the exit list. If there are addresses on it, the first thread is revived to execute them. The OS/2 kernel goes down the list and invokes each procedure in turn. When a procedure is entered, *SS* and *SP* are set to the top of the stack space defined for the process. *DS* is set to the process *DGROUP* if one was defined. (When the exit routine is located in a dynamic link package, it is not clear whether DS will address the *DGROUP* defined for the client process or the one defined within the package. It is safest to assume that *DS* must be explicitly loaded.)

An exit routine has the job of cleaning up whatever permanent resources it knows about. It can close files, clear semaphores, purge queues, and so forth. It does not have to worry about asynchronous events within the process, since only the one thread is alive. Unfortunately it gets no information on why the process is terminating. It can't know if the termination was orderly and voluntary, or whether it was unexpected and caused by an internal trap or an external signal. It can only clean up the global resources that it can, as best it can.

When an exit routine finishes its work, it does *not* exit by returning to its caller. It must use the *DosExitList* call to terminate; there is no valid return address on the stack. A return will typically cause a hardware trap of some kind. A hardware trap during exit processing does not cause a repetition of exit processing; it causes OS/2 to give up on the process entirely.

Reference: DGROUP 3.2.4.5

5.4 The Message Services

OS/2 contains three services to simplify the management of program messages. These have effect both at execution time and during development.

5.4.1 The Problem of Messages

Every serious application program has a long list of conditions it can diagnose, and an equally long list of messages to inform its user of them. It's important to have lots of clear, grammatical messages, but when they are embedded in the program several problems arise. Messages are bulky and inflate the size of the program; worse, most messages go unused during any single run of the program, so they take up space for no purpose.

There's another good reason not to embed messages in the program: it makes them hard to change. It's often desirable to edit messages for minor wording changes. If the messages are distributed throughout a large program, it can be a chore just to find the one that needs changing. Even when messages are centralized in a single module, it's a pain to have to do a compile and link for an editorial change, and it is also a waste of talent when a programmer has to be found to carry out a job that could be done by an editor, manager, or clerk.

Finally, a serious programmer must at least consider the possibility that the program will be used in a country where they speak a language other than the programmer's own. Even in the cozy, parochial United States there are significant markets for software that speaks French and Spanish. In other parts of the world it is taken as a matter of course that a successful program must serve speakers of several languages. The programmer may not be the one who translates the messages to another language, but the programmer has to plan how to make the translator's job possible.

5.4.2 The OS/2 Solution

One simple move begins to solve all these problems. The message text must be separated from the program—brought right out into a separate file that can be edited independently of the program source. That makes it easy for anyone, especially a nonprogrammer, to edit the messages. It takes their bulk out of the executable program file. It opens the way to wholesale replacement of the message file with one in another language.

Of course, there has to be a way of binding the program to its messages at run time. OS/2 supplies a system call that will look up a message in an external file. It also supplies a way to bind frequently-used messages to the program in advance, so that they can be kept and edited separately but still displayed quickly and without file I/O.

5.4.3 The Message Format

In order to simplify the message system, the OS/2 designers imposed a standard format on messages. The standard form is:

```
CCCnnnnX: The text...
```

In this format, the initial three letters *CCC* are a *component code* that should be chosen to distinguish all the messages from one major software function. All the messages issued by OS/2 itself, for instance, have the component code *DOS*. The idea is that anyone should be able to identify the source of a message at a glance from the component code. (This may be of most benefit to people who give technical assistance over the telephone. "Read me the message ... oh, that's not from our program, all of our messages begin with *QWE*.")

The component code must always be followed by a four-digit decimal number. The idea here is that four digits are enough to allow a unique number for every message in a large package—even to allow separate ranges for each subcomponent—while not overstraining the user's memory.

Finally, every message has a *classification* letter, indicated in the example by the letter *X*. The classifications that OS/2 recognizes are

- *I* for information
- *P* for a prompt
- *H* for help
- *W* for a warning
- *E* for an error diagnostic

Actually, the system calls used at execution time only recognize two classes: errors and warnings are in one, and all other classes are in the other.

5.4.4 The Message Text File

All the messages for an application are stored in editable files. Ordinarily there will only be one file, but there could be more of them. The rather rigid layout of the file is determined by the rules of the MkMsgf utility distributed with the system. It specifies that the file must contain

- Initial comments, marked by semicolons in column 1
- A line with the component code alone in columns 1-3
- Messages in the form *CCCnnnnX: Text...*

Each message begins in the first column of a new line. The appearance of a component code in columns 1-3 is what marks the start of a new message. As a result, messages may run for multiple lines. All the text from start of a message to the start of the next one are part of the message, including newlines. (That's

why there may not be comments in the body of the file; they'd be taken as parts of multi-line messages.)

5.4.5 The Binary Message File

The MkMsgf utility processes a message text file into a message binary file. The binary file contains the message texts, but it is prefixed by a table that relates message numbers, in binary, to the starting points of the corresponding messages in the file. This format allows a message to be looked up by number and loaded into storage quickly, without scanning the whole file.

The MkMsgf utility performs a certain amount of processing on the messages as it makes the binary file. It strips the *CCCnnnn* prefix from all Prompt, Information, and Help messages. Only messages classified Error and Warning retain their component codes and message numbers for display.

Wherever MkMsgf sees the marker *%0* (percent and zero), it stops copying text and skips to the next message. This feature is presented as a means of omitting the trailing newline from a prompt or any other message that should not be displayed with a newline. But because of the way it is implemented, it allows a way of embedding comments in the message text file. If you are determined to have them, just start every comment line with *%0* in column 1.

5.4.6 Displaying Messages

The *DosGetMessage* system call retrieves a message at execution time. Its primary inputs are a message number and the character string name of the binary message file, just as it would be presented to *DosOpen*. The system service opens the file (if it hasn't been opened already), looks up the message text, and copies it into a buffer supplied by the program.

The *DosGetMessage* call adds an important service to its basic purpose of retrieving messages. It also accepts a table of one to nine pointers to character strings that are to be inserted into the message. As it copies the message text to the buffer, it scans for insertion points that are marked by "*%d*" where *d* is a digit. It copies the insertion string from position *d* in the table at that point.

This service of copying with substitution is available separately as the *DosInsMessage* service. It takes a template and a table of insertion strings, and copies them to a buffer; however the program must procure the message text on its own.

The message services include one other service, mostly as a convenience. *DosPutMessage* is really a variant of the *DosWrite* file service. It takes what it presumes is an ASCII string and writes it to a file handle. That's just what *DosWrite* does. What *DosPutMessage* adds is simply that it inserts newlines as required to fit the message on a screen (or printer) that is 80 characters wide. It tries to break the display between words near the width of the screen.

The three services together make a very convenient, simple package for message display: call *DosGetMessage* with a number and a buffer, get back a buffer and a length and hand them straight to *DosPutMessage* for display on the standard output handle.

A few things are lacking from these services. Although the classification of a message is retained in the binary file, it is not returned by *DosGetMessage*—so the program can't find out whether the message it retrieved was classed as a Prompt or a Warning. Also lacking is the ability to format binary values for insertion into a message template, something like the invaluable *printf()* function of C. However, it would not be difficult to supply binary formatting as a layer of code on top of the *DosGetMessage* call.

Reference: *DosWrite* 8.11.3

5.4.7 Binding Frequent Messages

The MsgBind utility reads a control file, a message binary file written by MkMsgf, and the executable file of a program. It writes a new program executable that contains a selection of message texts as part of a code segment in the program.

When the program uses the *DosGetMessage* call, the support code checks for these embedded message texts first, before trying to find the message binary file. If the needed message has been bound to the program, the file operation is skipped. This takes care of one possible objection to the use of the message services: that they will slow the program down. Messages used frequently can be bound to the program and found rapidly; messages displayed rarely can be left in the file.

Once message texts have been bound to the program, they are part of its executable image and will supersede any changes in the message text file. The binding should be redone whenever the message binary file is updated (this kind of

thing can, of course, be automated with the Make utility). It is important to note that the MsgBind utility selects messages strictly by number, so it isn't sensitive to translations; as long as the right numbers appear in the binary file, the messages may be in any language.

The use of MsgBind is an optional step. If messages are never bound to the program, they will still be found and loaded from the file. Furthermore, a MsgBind run is easy; it is nowhere near as complex as running the Linker, and it doesn't need access to object libraries. It can be done by a nontechnical person after the program has been prepared. It could even be left up to the user of the program, who would be given instructions to copy the message binary file for the preferred language and then (if desired) to run MsgBind for better performance.

6

Managing Subordinate Programs

In this chapter we will examine how processes and sessions are created and managed.

Recall the three levels of OS/2. The *screen group* represents a logical screen, keyboard, and mouse. There can be more than one screen group, and the operator switches among them. The one that the operator is currently using is the *foreground* screen group. A screen group is used, or inhabited, by one or more processes. The screen group with all its inhabitants is called a *session*.

A *process* is one instance of a loaded program, plus an address space to contain its segments and the segments it has linked to dynamically, plus any other resources it owns, notably open files. A process is usually associated with a particular screen group, and uses the keyboard, mouse and video display of that screen group. Other chapters cover how a single process starts, ends, and deals with unexpected events. A process is animated by at least one *thread* of execution, possibly many threads. Another chapter covers how threads start, end, and coordinate their use of common resources.

In this chapter we discuss how one process may cause other programs to be loaded and run as subordinate processes, both in its session and in sessions of their own. We also take a cursory look at some services that are available only to a particular favored process, the top-level Session Manager.

References: screen group 1.2.1; session 1.2.2; process 1.2.3; program startup 4.3; signals and interrupts 5.3; threads 9.1

6.1 Subordinate Processes

One process may start others. A great deal of power is concealed in that simple phrase. It means that one program may command the services of any other program, or programs. It isn't hard to do, and the uses are endless.

6.1.1 The Process Family Tree

When a process creates another process, it creates an entirely new address space that contains another instance of a loaded program and another set of resources including open files. The new process will have at least one thread to execute it, and is free to start more threads. The new process is free to create yet more processes.

When one process creates another, the second is called a child of the first. An entire family tree of processes can be created when an ancestral process creates a first generation of children, and those create grandchildren, and so forth. That's what happens when OS/2 starts. The program named in the *protshell=* configuration statement is started as the founding ancestor of the family. On user command it starts other programs. Some of these are "shells" (command interpreters) in their own right, and they start other programs on user command. Those can start more processes, including new copies of shell programs. . . the mind boggles.

Reference: process creation and startup 4.3

6.1.2 Creating a Process

The fundamental service is the *DosExecPgm* call. It takes as parameters the name of an executable file and the two blocks of strings that go to make an environment segment. It creates a new process based on these; the new process also inherits all open, inheritable file handles belonging to the parent process. Since the parent process can control the contents of the environment and the meaning of all open file handles (including the standard handles), it has a great deal of control over the child program's operations.

References: opening inheritable files 8.7.3.3; child file handler 6.1.4

6.1.2.1 Modes Of Child Operation

DosExecPgm takes an option parameter that establishes the conditions under which the child process will run. In the Disk Appendix to this book there's a test driver program that allows you to experiment with starting child processes in different modes.

6.1.2.1.1 Synchronous Children

Option zero is the synchronous option. The child process is started, and it runs to completion. The *DosExecPgm* call does not return to the parent until the child has terminated. Then it returns an indication of why the child ended, and the child's exit code.

This is the only kind of child-creation that was possible in MS-DOS, and it is quite useful. It works in any situation where the child program is being used as a kind of subroutine, and the parent's work cannot continue until the child has finished. The traditional example has the parent creating a file of data, then invoking a sort utility as a child to sort the file. The sort utility serves the function of a subroutine of the main program. In a sense, it is a dynamically-linked subroutine, one that is bound even later and even more loosely than a normal dynamic link package.

This is also the way that a program might "shell out" to a command interpreter. That trick—invoking COMMAND.COM as a child process—is a common feature of MS-DOS applications. It's not hard to do in OS/2, either; it is easy to invoke CMD.EXE as a child. Shelling-out to DOS was common in MS-DOS because that was the only way the operator could get access to DOS commands, short of ending the application. In OS/2, the operator can create a new, concurrent command line at any time, so there's less need for it.

6.1.2.1.2 Asynchronous Children

Option number 2 of *DosExecPgm* creates a new process and returns to the parent at once. The thread(s) of the child process continue to run concurrently with the thread(s) of the parent and all the other processes in the machine (OS/2 shares the use of the CPU among threads, not processes as such). When the child finally terminates, OS/2 will save its termination and exit codes. The parent process can discover what they were by calling *DosCWait*.

This option lets the parent continue executing concurrently with the child. It has obvious uses for a command shell. In order to build a pipeline of programs, for example, the command shell has to get control back from starting one program in order to start the next one in the pipeline. Examples of its use to ordinary applications are harder to come by.

DosExecPgm also supports an option 1 that has the same effect, only the child's termination codes are not saved when it ends. As a result, the parent can wait for the child to complete but cannot find out why or how it ended. The only motive for this feature seems to be that it saves the OS/2 kernel an amount of storage that couldn't exceed a kilobyte, over what is probably a trivial span of time between the child's ending and the parent's query with *DosCWait*.

6.1.2.1.3 Detached Children

Another option of *DosExecPgm*, option 4, requests that the child process be detached from the parent's screen group. The new process isn't properly a child at all; the parent can't wait for its completion. Moreover, the new child is detached from any screen group. It can't use normal keyboard, mouse and screen access, although it can use the pop-up display screen group. A detached program may still use the standard input, output and error files provided those handles are associated with disk files, not with the screen and keyboard it doesn't have.

This is the basis of the Detach command of the OS/2 command shell. It starts a process that doesn't need to interact with the operator, usually some kind of system monitor like a print spooler.

6.1.2.1.4 The Debugger's Target

Yet another *DosExecPgm* option, number 3, loads the child process and returns its PID; the child is suspended pending a *DosPTrace* call. The *DosPTrace* call permits the parent process to inspect and patch the contents of the child's address space, and to control the execution of the child's threads.

The building of a debugger is a delicate, complex subject. The *DosPTrace* call is very complicated, but only sketchily documented in the manufacturer's literature, and time does not permit me to explore its operation by experiment, at least for the first edition of this book. Accordingly, *DosPTrace* and debugging in general are not documented further in this book. If this lack causes you serious concern, see the notice on how to communicate with the author in the Preface.

Reference: process ID 5.1.3

6.1.2.1.5 Suspended Children

Finally, option number 5 of of *DosExecPgm* creates the new process but does not allow it to start executing. The process is "frozen" until it is "thawed" with a call to *DosSystemService*. This enables the parent to create the child and then switch it into a different screen group before it begins to execute. Unfortunately, *DosSystemService* isn't available to any but that ancestral shell program. Since any other process has no way of "thawing" a child, it had best not create one using option 5.

6.1.3 The Child's Environment

Two of the *DosExecPgm* parameters combine to create the environment segment that the new process will receive. The first is a far pointer to a block of environment strings. This parameter may be passed as a null pointer (a doubleword of zero) and *DosExecPgm* will copy the environment of the parent process. But the parent has the opportunity of making its own copy of the environment, modifying it in any way. New named variables could be inserted. Existing variables could be deleted or given different values. For example, the parent can set any PATH variable it likes, and thereby either limit or enhance the child's ability to start children of its own. The child process has no choice but to accept the environment variables it inherits; there is no mechanism by which it could gain access to its parent's environment.

The other environment parameter is a far pointer to the second block of strings, the argument strings, that go into an environment segment. All compiled programs expect to find at least one string in this block, a string containing the name of the executed program. Since the identical string has to be passed to *DosExecPgm* as the file specification of the program to load, there's no reason not to copy it as the first argument string.

The second string in the argument block is the one that compiled programs will treat as the "command tail." A C program will parse it into the child's *argv* list, and a Microsoft Pascal program will interpret it to assign values to program parameters. Unfortunately, there's no command shell to scan this string for redirection or "pipeline" symbols; it will be seen only by the child program.

Programs that ordinarily execute from the command line won't expect more than two strings in the argument block. One might imagine all sorts of schemes for passing data in this space, but it is probably simplest and best to pass additional parameters in the form of named variables in the environment block. These can be set up by a parent or manually from the command line. A program that looks for special argument strings can only succeed under its designed parent.

Reference: environment segment use 5.1.5

6.1.4 The Child's File Handles

The parent process determines what open files its children will inherit. In general a child inherits all the files that were open for its parent. In other words, a parent's active file handles become implicit parameters of a child program. Programs that can execute from the command line expect only three handles to be open: standard input (zero), standard output (1), and standard error (2).

Reference: file handles 8.4

6.1.4.1 Passing Explicit Open Handles

A program designed to be invoked only as a child of a certain parent might be written to expect, say, handle number 8 to be open to a special file. However, such a program could never be invoked from the command line since the command shell would not set up the needed file. It is generally a better plan to pass a child the names of its files as command parameters; then the child can be executed as a command.

6.1.4.2 Preventing Inheritance

There are good reasons to *prevent* a child from inheriting the parent's open files. One is that the active handles fill up slots in the child's table of files. That may cause the child to get the error "no available file handles" when opening a file of its own. Another reason is that a child's handle has exactly the same power over a file as the parent's. A wayward child program can use a file incorrectly, and the parent can do nothing to prevent it.

When a file is opened, an inheritance flag is specified as one of the options of *DosOpen*. If the flag is set for no inheritance, the handle for that open file is not

inheritable. As far as a child process can tell, that handle is just an available, unused number. However, "do allow inheritance" is the default option that will usually be selected by a language's library routine for opening files.

The *DosSetFHandState* call can be used to change the inheritability of a handle. It should be used as needed to prevent a child process from inheriting irrelevant handles. It could be used to change a handle so that one child process inherited it and another did not.

References: *DosOpen* mode word 8.7.3; *DosSetFHandState* 8.10.1

6.1.4.3 Preparing Standard Files

Many useful programs take their input from the standard input handle and write their output to standard output. It's common to call such a "filter" program as a child, first setting up the standard handles zero and one so that it will take its input and leave its output where the parent wants.

The main tool for this is the *DosDupHandle* call. It has two functions: it can coin a new handle which refers to an existing, open file, or it can close an existing handle and then make it refer to a different, open file. Both functions are used to prepare a standard file for a child process.

Here's how it's done. A file handle exists that is to be passed to a child as its standard input. Let's suppose it is saved in a variable *h*. It should not be inheritable; if it was opened as inheritable *DosSetFHandState* should be used to change it.

The first step is to use *DosDupHandle* to coin a new handle for the present standard input file. That preserves the parent's access to that file. Let's suppose that the new handle is saved in a variable *n*. The new handle is inheritable, and *DosSetFHandState* should be used to make it uninheritable so that it won't clutter up the child's file table.

Next, *DosDupHandle* is used to make handle zero identical to handle *h*, the file the child is to get as standard input. Now the child program may be started; it will inherit the parent's handle zero, which refers to the same file as handle *h*.

Immediately after starting the child process, the parent can use *DosDupHandle* again to make the parent's handle zero identical to handle *n*. That restores the

parent's standard input. When that has been done, handle *n* should be closed in order to make its number available.

The same steps could be used to prepare a replacement for the standard output or standard error files.

Reference: table of file handles 8.4

6.1.5 Communicating with Pipes

A *pipe* is a buffered path for data. It is used like a sequential file; data is put into the pipe with *DosWrite* and taken out with *DosRead*. Between writing and reading, the data in the pipe are held in a buffer in storage. The buffer is known only to OS/2; the processes that use the pipe know it only by its handles.

A pipe is a one-way path for data. It has a read handle and a different write handle. The read handle appears to be an open, read-only, sequential file. The write handle appears to be an open, write-only, sequential file.

Ordinarily a pipe is used to send data from one process to another; however, it could be used by two threads of the same process. One thread writes; the other reads. The buffer in the middle means that they don't have to synchronize their activities very closely. So long as there is room in the buffer, a write is instantaneous. If the buffer is full, the thread trying to write will be suspended until some other thread does a read, making space available. If the reading thread finds the buffer empty, it is suspended until a write is done.

Reference: *DosRead* and *DosWrite* 8.11

6.1.5.1 Pipe Handles

The *DosMakePipe* call creates a pipe and returns its two handles. The handles are inheritable. Initially only one handle refers to each end of the pipe. However, handles may be duplicated, either deliberately with *DosDupHandle* or automatically due to inheritance, and handles may be closed, either deliberately or by the termination of a process.

OS/2 keeps track of how many handles refer to each end of the pipe. When all the handles for one end of a pipe have been closed, the pipe is said to be "broken." If all the write handles are closed, no more data will be entering the

pipe. After all buffered data has been read, an attempt to read the pipe will return a data length of zero, the end of file signal.

If all the read handles to a pipe are closed, there will be no more reading done, and further writing would be a waste of time. In this case an attempt to write to the pipe yields an error code.

6.1.5.2 Passing Pipes to Children

A pipe's read or write handle is a file handle. A child process can be given the reading end of a pipe as its standard input, or the writing end of one as its standard output or standard error file. The techniques of using *DosDupHandle* and *DosSetFHandState*, discussed above, work identically when the handle to be passed is a pipe.

One additional step is recommended, however. After starting the child process, the parent should close its own copies of the handles the child inherited. That ensures that, when the child terminates, the pipe will be broken. Here is the complete plan for passing a pipe to a child as its standard input.

- Coin a duplicate of handle zero; call this *n*.
- Make *n* uninheritable.
- Create a pipe, producing read handle *r* and write handle *w*
- Make zero be a duplicate of *r*.
- Make *r* and *w* uninheritable.
- Start the child process.
- Make zero be a duplicate of *n*.
- Close *n*.
- Close *r*.

Now the child has the only read handle of the pipe as its standard input. The parent has the only write handle stored as *w*; it can be used to feed input to the child.

The child's handle zero is the only handle to the reading end of the pipe. When the child terminates, the pipe will be broken and an attempt to write will produce an error—not the endless wait that would result if the parent retained handle *r*.

A very similar process can be used to give the child a pipe handle as its standard output. Replace "zero" with "1" in the plan above, and reverse the roles of *r* and *w*.

The startup routine of a compiled program generally tests its standard input file with *DosQFHandType* to see if it is a device or a file. If standard input is a pipe, the program will discover only that it is not a device. Since it is not a device, the runtime library routines for standard input will assume it to be a file. Therefore they will usually expect every input line to be ended with a carriage-return and linefeed byte pair. (From a device, they would take either return or linefeed as end of line.) If a child process seems to hang while reading from a standard input pipe, or processes the input strangely, make sure that the data going into the pipe has two-byte line delimiters.

The *DosRead* system call, when applied to a pipe, will wait only if the pipe buffer is completely empty. If there is any data in the pipe, *DosRead* takes as much as will fit in its buffer (usually it can empty the pipe) and returns. However, that may not be a complete data record. For example, if a child program written in Pascal is writing into a pipe as standard output with a statement like this:

```
writeln(j:8,'Field  2',k:8);
```

each of the three fields of the output line will be put into the pipe as a separate *DosWrite* call. The parent's call to *DosRead* is quite likely to return with only the first field, or only the first two fields.

When reading delimited data, be careful to keep reading until the delimiter appears. When reading fixed-length records, keep reading until the length has been accumulated. In either case, the *DosRead* call will suspend until data is available. If the child terminates unexpectedly, the pipe will be broken (provided that the plan sketched above has been followed) and the *DosRead* call will end with a data length of zero.

In the Disk Appendix to this book there is a test program that starts a child process with a pipe for both its standard input and standard output.

References: *DosQFHandState* 8.10.1; *DosRead* 8.11.1

6.2 Subordinate Sessions

Sessions are created and switched by a top-level process, the Session Manager. OS/2 users are familiar with the Session Manager's two-panel menu, the left panel headed "Start a program" and the right, "Switch to a running program."

Whenever a selection is made from the left panel, the Session Manager creates a new session, composed of a new screen group and a new process built from the selected program. We may suppose that it uses *DosExecPgm*, probably with option 2, to start the process. If you have used the Session Manager's "Add a program title" function, you will have noticed that the two of the fields it asks for are exactly the pathname and argument strings that are needed with *DosExecPgm*. It must pass the null string for the environment, which explains why the command shell always starts with an empty environment.

It is also the Session Manager that monitors the operator's keystrokes and switches the hardware between screen groups in response to the Alt-Escape and Control-Escape hot keys.

The Session Manager makes these functions—creating new sessions and switching between sessions—available to application programs through four system calls.

This is an expansion of the *DosExecPgm* service to a level of function that's unique in my experience. When we start a child process with *DosExecPgm*, we don't usually expect it to interact with the user; that's ordinarily the job of the parent process. Or if the child program is interactive, it would be started with option zero, suspending the parent process until the child had finished. That way there would be no contention for the use of the screen, keyboard and mouse.

With the services of the Session Manager, we can start child processes that can interact with the user through a completely separate set of hardware. We can start a child process and then wear it like a mask, so the operator sees only its session and the parent session is hidden. Or a parent process can start several child sessions and select which will be visible and when, in effect becoming a subordinate Session Manager in its own right.

In the Disk Appendix to this book there is a test driver program with which you can experiment with these facilities from the keyboard.

6.2.1 Starting a Session

The *DosStartSession* call initiates a new process in a new screen group. It takes its parameters in the data structure shown in Figure 6.1.

The last four fields of this structure are far pointers to ASCIIZ strings. Three of these strings serve identical functions to the three fields of the Session Manager's

"Add a program title" window. The first specifies the title that the Session Manager will display in its "Switch to a running program" menu. The second must be the fully-qualified drive and pathname string of the program executable file. And the third is the string that the program will receive as a "command tail" in its environment segment. We may suppose that these last two strings will be passed straight into a *DosExecPgm* call.

The last field of Figure 6.1 may be a doubleword of zero, or it may be a far pointer to a string that is the name of a queue created with *DosCreateQueue*. In some circumstances the Session Manager will post an item to that queue when the new screen group terminates.

A session consists of a screen group, the top-level process in the screen group, and any child processes it may start. The session ends when the last process inhabiting the screen group terminates. For example, a session with CMD.EXE ends when the Exit command is entered. The screen group is dismantled.

If a session is in the foreground when it ends, what happens next depends on how the session was started. Either the Session Manager's screen or the parent session's screen will pop up in its place, depending on circumstances.

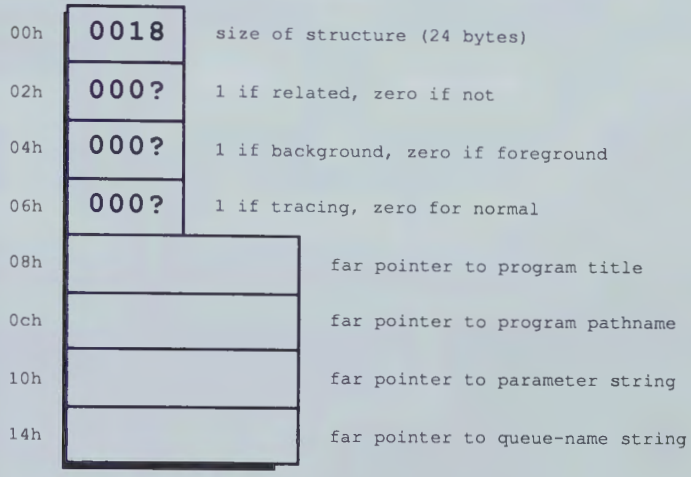
6.2.1.1 Foreground and Background

When the third field of Figure 6.1 is zero, and provided that the parent process is in the foreground, the new session will start in the foreground, displacing the parent. When the field is 1, the new session will start in the background, invisible. The user will have to switch to it with a hot key in order to see it.

6.2.1.2 Unrelated Sessions

When the second field of Figure 6.1 is zero, the new session is unrelated to the caller's session. The *DosStartSession* call returns nothing but an error code; if that's zero, the new session has begun. The caller has no way of finding out when the new session ends, if it ever does. It is simply spun off to be used by the operator exactly as if the operator had selected it from the Session Manager's "Start a program" menu.

Figure 6.1
Data Structure Passed to DosStartSession



Regardless of whether the new session is related or not, the process in it is always the child of the Session Manager. The calling process is never the parent and can't wait on it with *DosCWait*.

If an unrelated session ends while in the foreground, the Session Manager's screen pops up.

6.2.1.3 Related Sessions

When the second field is 1, the new session is related to the caller's session. The Session Manager returns the ID of the new process; therefore the parent will be able to signal it with *DosFlagProcess*, give it storage segments with *DosGiveSeg* and try to end it with *DosKillProcess*.

The Session Manager also returns a "session number." This appears to be a fairly large binary number, most likely an offset in some private data segment. The number can be used to control the child session in certain ways.

If a queue-name is specified when starting a child process, the Session Manager notes it. When the session terminates, the Session Manager opens the queue with *DosOpenQueue* and puts an item in it with *DosWriteQueue*. The item con-

tains a far pointer that addresses two words. The first is the session number, the other is the exit code of the child process.

If a related session ends while in the foreground, the parent session pops up in its place.

Reference: queues 9.7

6.2.1.4 Making a Command-Line Session

For an easy example of how a related session might be used, consider creating a command line from inside an application, the way MS-DOS applications "shell out to DOS" to let the user do a few commands. But instead of just starting CMD.EXE as a subroutine child process, start it in its own screen group. Then the parent process can go on working, including screen output, while the user works with the command line interpreter.

The program title would be something descriptive like "Temporary Command Line." The program pathname is usually available in the *COMSPEC*= environment variable as in MS-DOS.

For a program parameter string would be something like "/k echo type 'exit' to return." The "/k" switch tells CMD.EXE that a command follows and that it is to prompt for more after executing it. The user will see the injunction "type 'exit' to return" as the first line on the new screen. When the user does give the EXIT command, the child session will end and the parent session will pop up in its place. The parent process can discover this by periodically checking its termination queue for the item posted by the Session Manager. "Ah, you're back," it might say to the user.

6.2.2 Switching and Binding

There is a *DosSelectSession* call that will bring a child session to the foreground to replace the parent session. Its effect is temporary. However, a more intriguing service provides for "binding" a child session to the parent, so that whenever the operator selects the parent session, the child pops up instead. It's almost as if the parent was wearing the child session like a mask; the parent session simply cannot be brought to the foreground until the binding is ended.

Binding is done with the *DosSetSession* call. The parent can "bind" any one child session at a time. If the parent session is in the foreground when the call is executed, it instantly vanishes and the bound child pops up in its place. Binding a different child breaks the bond with the first. If a bound session ends, the binding is broken as well.

The *DosSetSession* call may also be used to make a child session unselectable. An unselectable session will not come up in the normal rotation of sessions with Alt-Escape. It appears in the Session Manager's menu, but can't be selected from it. The session exists but will never come to the foreground.

At least, it won't come to the foreground in its own right. It will come to the foreground if it is bound to the parent session and the parent session is selected. This means that a parent could create several child sessions, make all of them unselectable, and then make one or another visible by binding it.

In order to do this, the parent would have to receive some kind of operator signal. The signal might come from some special hardware, but the usual channel would be the keyboard or the mouse. But that's not possible; the keyboard and mouse go with the screen group, and when a child session is in the foreground, it receives keyboard and mouse input. Intriguing puzzles remain to be solved in this area.

6.3 Restricted System Calls

The Session Manager has a privileged place in the OS/2 system. It is a process (usually process ID 3) and uses OS/2 services to do its work. However, there are some OS/2 kernel services that are designed to support the Session Manager and be used only by it. Although these system calls can be linked, they will return an error code to any calling process except the Session Manager. This is enforced by an interesting method: the very first process to issue one of these calls is presumed to be the privileged user, and its process ID is noted. Calls from any other process are rejected. Since the Session Manager is executed from the *protshell=* statement of the configuration file, long before any other program, it is sure to get the opportunity to make these calls first.

Here are the names of the system calls reserved to the Session Manager, so far as I've been able to determine them. They don't appear in the reference section of this book. There may be others.

- *DosSGSwitch*—bring a session to the foreground
- *DosSGSwitchMe*—bring caller's session to foreground
- *DosSGSWitchProc*—switch process to another session?
- *DosSetFG*—make process favored for dispatching?
- *DosSystemService*—miscellaneous special functions

6.4 Data Into Code

There are applications that create machine code on the fly, and then want to execute it. There are quick-response language systems that compile into storage and then execute the compiled code. And on-the-fly compilation is a technique used by sophisticated interpreters of such languages as APL and Prolog, and by some sophisticated database managers.

The CPU hardware draws a sharp line between code and data. A segment of storage is either code or data; if it's code, the program can't modify it, and if it's modifiable it must be data and data can't be executed as code.

The *DosCSAlias* call is designed for just this situation. It takes the selector of a data segment and creates an extra descriptor for that segment in the LDT of the calling process. The new descriptor calls the segment code. The new selector can be used in an indirect far call to call the contents of the segment as code.

The new descriptor has the "conformable" attribute, which means that the segment takes on the privilege level of its caller. Therefore the generated code may be called from an ordinary segment or from one with I/O privilege. If the caller has I/O privilege, the generated code may contain I/O instructions.

Reference: comformable segments 2.5.2

7

Storage Allocation

OS/2 manages most of storage as a collection of blocks that it can allocate, move or swap to disk. The storage devoted to real-mode programs (below the *rmsize* boundary specified in the configuration file) is a single block that is managed by the DOS system that runs in it. Hardware designates other blocks as video buffers and firmware. But writable storage above *rmsize* is available for allocation by the services described below.

7.1 Moving Segments

Protect-mode programs never know the physical address of the segments they use. Therefore OS/2 is free to move segments in order to bring unused areas together into large, contiguous blocks. This may be forbidden by *memman=nomove* in the configuration file. Normally it is not, and when OS/2 is asked to allocate a segment larger than the largest vacancy, it will move occupied segments as necessary to open up a vacancy of required size.

7.1.1 Performance Impact

When segment moving is allowed, a request to allocate storage will never fail for lack of space unless there really is not that much space anywhere in the system. However, the system might possibly spend many milliseconds shuffling segments around to make room. The actual delay will usually be very small, but the amount is unpredictable and may be large. Normal applications don't care, but some dedicated applications may not be able to tolerate an unpredictable delay. That's why it's possible to disable segment moving. When segment moving is

disabled, there is a chance that a request to allocate storage may fail for lack of space even when enough space is available in the aggregate.

7.2 Swapping and Discarding Segments

Besides moving segments, OS/2 can use disk space as an extension of physical memory. The technique is called *swapping* in the OS/2 manuals; it is done unless forbidden by *memman=noswap* in the configuration file.

Swapping is done only when necessary, that is, when OS/2 is asked to allocate a segment that is larger than the available vacant memory. In fact, when swapping is allowed, OS/2 can make space available for allocation up to the size of physical memory less a reserve. (The amount of the reserve is discussed below.)

7.2.1 Discardable Segments

When there isn't room to allocate, OS/2 tries first to *discard* segments. It may discard any read-only segment for which a reliable copy exists on disk, which is to say any code segment of a program or dynamic link library that was loaded from a fixed disk. Some data segments are allocated with a "disposable" option and these too may be discarded.

When it discards a segment, OS/2 turns off the "present" bit in the segment's descriptor in every descriptor table in which it appears. That may be just one table for a segment that is part of only one process's address space, or many tables for a code segment of a dynamic link library. When a process tries to refer to that segment a hardware trap will occur and OS/2 will reload it. In the meantime the storage that it occupied becomes free for allocation.

There will normally be many discardable segments. OS/2 chooses to discard the segment that has gone unreferenced the longest. It learns this with the help of the CPU hardware. The CPU sets the "accessed" bit in the descriptor of every segment that is addressed; that is, it sets the accessed bit when a selector for that segment is loaded into a segment register.

OS/2 keeps a list of all segments that might be discarded. At the end of each timeslice, it scans the local descriptor table of the process that ran during that timeslice. Where it finds the accessed bit set, it moves that segment to the head of the list of discardable segments, and clears the accessed bit. Thus the segments

at the tail of this list were used least recently, and these are the segments that are discarded first when discarding is necessary.

References: present and accessed bits 2.4.3.3; allocating discardable segments 7.3.7

7.2.2 Swapping Segments Out

When all discardable segments have been discarded and there still isn't enough room for allocation, OS/2 creates more discardable segments by writing data segments to the swap file. Once a segment has been copied to disk, it needn't be retained in storage.

Only entire segments are written to disk and discarded; OS/2 does not *page*, that is, swap fixed-size pages between storage and disk. It is not possible to allocate segments in such a way that they can't be swapped and discarded. It is possible for I/O device drivers to temporarily lock individual segments in storage so that they can be neither moved nor swapped, but in general the only way to ensure that segments stay in storage at all times is to disable swapping.

When swapping is not allowed, the largest segment that may be allocated at one time is limited by the amount of vacant space at that time. When swapping is allowed, OS/2 can create vacant space up to the size of physical memory less a reserve. The reserve is four times 64K bytes, the most that one program can reference at any instant using the segment registers of the CPU. If it allocated a segment that filled storage without leaving that reserve, OS/2 could find itself in a situation where no program could run. With the reserve, any program can be made to run, although in the worst case it might run very, very slowly due to constant swap I/O.

7.2.3 Swapping and Performance

When swapping is allowed, the potential delay during allocation is quite large. The process that requests allocation might be suspended while one or more segments are being written to disk. Such delays should be rare, but it is impossible to predict when they will occur or how long they will be; these things depend on the whole state of the system and on the speed of the disk drive used for swapping and the other disk I/O that may compete with swap I/O.

Furthermore, swapping may cause unpredictable delays for other processes besides the one that allocated the storage (this was not the case with moving segments,

which delays only the process waiting for storage). After all, the segments that were discarded will probably be referenced at some (unpredictable) later time. Then there will be a hardware trap and OS/2 will have to recover the missing segment from the program file or the swap file.

Before it can recover a discarded segment, OS/2 has to allocate space in memory to receive it—and this may require it to discard other segments. In the worst case, OS/2 might have to write segments to the swap file in order to make room to read back a segment that was discarded. The process that suffered the *segment fault* (as the hardware trap is called) is suspended during the disk I/O, which will take an unpredictable time that depends on the speed of the drives involved and the amount of competing disk activity.

The delays inherent in swapping are compounded by the relatively slow fixed disks available in desktop computers (they are many times slower than the disks used for swapping in mainframe and minicomputers where the techniques were developed). Furthermore, swap disk I/O competes with file disk I/O to the detriment of both. (In mainframes that swap, the fastest disk drives are often dedicated to swap I/O so as to keep the two kinds of I/O separate.) If swap I/O exceeds one or two segments per second, the performance of an OS/2 system will probably be sharply degraded.

However, swapping does permit the system to overcommit physical storage. With swapping, an OS/2 user can load more and larger programs than would otherwise fit in storage. So long as the user doesn't expect to switch rapidly between these programs, and doesn't mind a delay for swapping segments in and out as each program is activated, swapping can be very convenient.

7.3 Storage Allocation

OS/2 provides convenient, flexible services for dynamic storage allocation. Your program can allocate segments in sizes from one paragraph to 64K bytes and so-called "huge" segments from 64K bytes up to the capacity of the machine. Once allocated, segments of any size may be grown or shrunk. Segments can be shared between processes in a variety of ways. There are services for suballocating space within un-huge segments (shared or not); these save you the trouble of devising your own memory-management scheme.

7.3.1 Querying Available Storage

The *DosMemAvail* call returns a 32-bit integer giving the size in bytes of the largest contiguous vacant block at the time of the call. In a system that permits neither moving or swapping, this number is useful; it is the largest block of storage that can be allocated. Of course, there is no guarantee that the number will remain accurate. Another process might request that block of space just one microsecond after your program receives the number.

When segment moving is allowed, *DosMemAvail* number is often too pessimistic; it doesn't reflect the aggregate amount of storage that OS/2 could assemble by moving segments around. When swapping is allowed, one segment up to the size of physical storage less 4*64K bytes can be allocated, and as many of these can be allocated as will fit on the swap device.

In short, *DosMemAvail* has mainly curiosity value. The best way to find out if storage is available is to try to allocate it.

7.3.2 Allocating Normal Segments

The *DosAllocSeg* call creates a storage segment and makes it part of the address space of the calling process. A descriptor for the new segment is installed in the local descriptor table of the process. The descriptor type is set to "data" (see the *DosCreateCSAlias* call for a way to execute code in a data segment).

Reference: *DosCreateCSAlias* 6.4

7.3.3 Allocating Huge Segments

The *DosAllocHuge* call is similar to *DosAllocSeg*. However, the architecture of the 80286 CPU permits segments to be no more than 64K bytes long. That is the limit of the segment-size field of a descriptor table entry. In order to give access to a contiguous block of greater size, OS/2 sets up as many descriptor table entries as necessary, each one except the last describing a 64K byte segment of the huge block.

Reference: descriptor table and segment descriptors 2.4

7.3.3.1 The "Huge Shift"

The descriptors are spaced at regular intervals (usually adjacent) in the local descriptor table of the process. The selector for the first (lowest address) descriptor is returned by the allocation call. The other selectors can be developed by arithmetic on the first one. The increment per 64K segment is a system constant that is available in two ways. It can be gotten with the *DosGetHugeShift* call, and it is a field in the system information record returned by *DosGetInfoSeg*.

The increment that steps from one huge selector to the next is encoded in an unusual way. It is not provided as a constant, but as a power of 2; the value returned by *DosGetHugeShift* is a shift count. If the word 0001h is shifted left by that amount, the result is the numerical difference between one huge selector and the next.

There are reasons for this odd encoding. One is that the shift need not only be applied to 0001; it can be applied to the ordinal index of any 64K segment in the huge allocation. To address the sixth segment of a huge block, shift the word 0005h left by the Huge Shift and add it to the base selector. If the increment were given as a constant, multiplication would be required.

Reference: global information segment 5.1.4.2

7.3.4 Allocation and the Local Descriptor Table

Whenever a segment is allocated, at least one entry in the local descriptor table of the process is used. There is a hardware limit of 8,192 descriptors, but OS/2 reserves a large (unspecified) number of these for addressing code segments, especially dynamic link segments.

It is quite possible for a process to allocate so many segments that it runs out of room in its local descriptor table. *This error is returned as code 8, out of memory*. Therefore it is possible to get code 8 when physical storage is still available and when another process with fewer segments in use would still be able to allocate. It is best to allocate the smallest number of segments possible and make each one as large as possible.

Reference: allocation of local descriptors 2.4.6

7.3.5 Suballocation

Often we need a way to carve a segment into smaller objects, and to allocate and free these smaller objects over and over. It is not difficult, in fact it's rather fun, to design one's own storage allocation scheme, but it can be a distraction from the main job. OS/2 offers a generalized solution for the problem.

First use *DosAllocSeg* to allocate a segment as a pool to hold all the small objects you plan to create. The pool can be resized later if necessary. It appears that there is an overhead of eight bytes for each suballocated object, so allow for that when computing the size of the pool. (OS/2 carefully refrains from specifying the details of suballocation, so this is only informed speculation. If you don't trust them, do it yourself.)

A call to *DosSubSet* initializes the pool. Then objects may be allocated within the pool by calling *DosSubAlloc*, and they may be freed by calling *DosSubFree*. The pool can be grown to the limit of 64K bytes or shrunk with another call to *DosSubSet*.

The *DosSubAlloc* and *DosSubFree* calls may be used asynchronously from concurrent threads in the same or different processes.

7.3.6 Resizing Allocated Space

Once a block of storage has been allocated, it can be resized. *DosReallocSeg* will grow or shrink an ordinary, nonhuge segment. However, if the segment is being suballocated, *DosSubSet* must be used to resize it.

DosReallocHuge can resize a huge segment. A huge segment may only be grown to a maximum size that was stipulated when it was allocated, but it may always be shrunk.

7.3.7 Disposable Segments

When any segment is allocated, an option may be used that makes it possible to mark the segment as disposable. Initially the option has no effect; the segment behaves like any other allocated data segment.

At some time the program may judge that the contents of the segment are no longer essential. They are not immediately needed and, for whatever reason, they

could easily be reproduced or read back from disk. If they are kept the program can use them later, but if they are lost the only cost will be some CPU time to rebuild them.

This is the point at which the program may call *DosUnLockSeg*. That call removes the selector from the descriptor table (it is no longer valid to address the segment) but does not release the segment as vacant space. It only marks the segment as discardable. Note the difference: the system treats code segments as discardable because it knows it can recover their contents from the program file. It treats swapped-out segments as discardable because it can recover them from the swap file. But it treats an unlocked segment as discardable because the program says the segment can somehow be recovered.

The unlocked segment may or may not be discarded to make room, depending on the activity in the system. Later, the program may decide that it needs the contents of the segment. It will call *DosLockSeg*, which looks to see if the segment was discarded. If it was not, the selector is restored in the descriptor table and the segment is once again part of the address space. However, if OS/2 had to discard the segment, *DosLockSeg* returns an error. The segment must be restored by reallocating its space and regenerating its contents.

Disposability of a segment is controlled by a reference count. The count is set to 1 when the segment is allocated. Each *DosLockSeg* call increments it; each *DosUnlockSeg* call decrements it. The segment is only made discardable when the count goes to zero. This method is used because a disposable segment may also be a shared segment, accessible from more than one process asynchronously. With the reference count, if one process locks the segment, another's unlock won't take effect until the first process also unlocks it.

Not many programs have the characteristics that make disposable segments worthwhile: quantities of data that are useful but not essential in storage buffers. (A file-browser would be the perfect example. It would read pages of a document from disk into segments for display, and after the user moves to other pages it would unlock the data segments. If the user returned them it could try to recover them; if they had been discarded it could read them from disk.) When the characteristics exist, the feature lets the program tune its behavior to the load on the system.

Reference: discarding segments 7.2.1

7.3.8 Freeing Storage

All storage blocks, however they were acquired, are released in the same way, by calling *DosFreeSeg*. This powerful function is used to release huge and nonhuge segments and shared segments of all kinds. A disposable segment can be freed even after it has been discarded, although the only advantage of doing so is that it reclaims a slot in the local descriptor table. *DosFreeSeg* can even be used to release code and data segments loaded with the program.

7.4 Sharing Allocated Segments

When it is first allocated, a segment is part of the address space of just one process. There are three distinct ways of sharing an allocated segment between processes, that is, making the segment part of more than one address space at a time.

Sharing is necessary in many cases. For instance, the queue operations can only pass four-byte data items through a queue itself. To pass larger items it is necessary to share storage between processes, place the items in shared storage, and pass the addresses of the items in the queue.

Once a segment is shared, more than one process may access, and potentially may modify, the contents of the shared segment. It becomes imperative to coordinate the use of the shared storage. Otherwise two processes may write over each other's work, or one may try to use data that the other has only half-prepared. The only practical tool for interprocess coordination is the semaphore.

The three methods of sharing are segment Naming, segment Getting, and segment Giving. In all three methods, the shared segment persists so long as at least one process has access to it. When all processes that had access have either released it with *DosFreeSeg* or simply terminated, the segment is released.

In only one method, Getting, do all processes use the identical selector number for the shared segment (so that far pointers into it are relevant in all contexts). In only one, Naming, can unrelated processes share a segment without first exchanging information to identify the segment.

References: queues 9.7; coordinating asynchronous processes 9.2: allocation of descriptors 2.4

7.4.1 Sharing Segments by Naming

The *DosAllocShrSeg* call creates a nonhuge segment much as *DosAllocSeg* does. The difference is that this call associates an ASCII name-string with the segment of storage. The name of a shared segment has the same format as a file pathname. All shared segments are presumed to be located in a directory named SHAREMEM, so "\SHAREMEM*name.ext*" is the form of the name string. This is similar to the naming scheme for system semaphores.

One process creates the segment; any number of other processes, in any order and after any amount of delay, may gain access to it simply by calling *DosGetShrSeg* with the name. That call takes the name string, locates the segment, installs a descriptor for it in the first available slot in the calling process's descriptor table, and returns the selector.

Each process knows the shared segment by a different selector (which is to say, OS/2 uses the first available descriptor in the local descriptor table of each process). When the processes exchange addresses of objects in the shared segment, they should only exchange the offset values; the selectors are useless in the context of another address space. If processes share more than one segment by this means, they will have to arrange some means of relating an offset within a segment to the segment to which it applies.

Reference: pathnames 8.1

7.4.2 Sharing Segments by Getting

When a segment is allocated with either *DosAllocSeg* or *DosAllocHuge*, an option may be used that makes the segment eligible for sharing with *DosGetSeg*. (This is a completely different sharing method than sharing with *DosGiveSeg*; the two calls are not related.) OS/2 chooses a selector for the new segment from among the selectors it reserves for segments that have to be known in more than one address space (dynamic link segments use selectors from this group).

At any later time, a different process may call *DosGetSeg*, passing the same selector. That selector will be activated in the calling process's address space; it will point to the shared segment. In this sharing method, the segment is addressed with the identical selector value in every process. Thus the far address of data in one of these segments is usable in every process.

This method of sharing requires that receiving processes know the selector number that was returned to the allocating process by *DosAllocSeg*. This information may be different every time the programs run, so it must be conveyed at the time of execution. There are not many ways of passing it. Between unrelated processes, the receiving process could create a queue and the allocating process could write the selector into it as a queue item. A parent process could convey it to a child process through a pipe or by converting it to ASCII and passing it as a command-line parameter or an environment variable.

References: queues 9.7; pipes between parent and child 6.1.5; child process environment 6.1.3

7.4.3 Sharing Segments by Giving

When a segment is allocated with either *DosAllocSeg* or *DosAllocHuge*, an option may be used that makes the segment eligible for sharing with *DosGiveSeg*. (This is a completely different sharing method than sharing with *DosGetSeg*; the two calls are not related.)

Later, the allocating process may use *DosGiveSeg* to install the segment in the address space of another process. It has to pass the process ID of the recipient. The only process ID easily determined is that of a child process, so this method of sharing is especially convenient between parent and child processes.

DosGiveSeg locates the segment in the local descriptor table of the creating process and copies the descriptor into an available slot in the receiving process's table. In the case of huge segments, all descriptors are copied, so the receiving process can access the whole block just as if it had itself called *DosAllocHuge*.

In this method, different processes know the storage by different selectors, so they should exchange only offsets (or offsets and ordinal segment numbers, in the case of huge segments). If they share more than one segment by this method they will have to work out some way of relating an offset to the segment in which it is relevant.

Reference: child processes 6.1.2

8

Files

The many file-related system calls may be divided into three groups: those used to manage the file system, those used to manage disk directories, and those that operate on single files.

The first group includes calls for getting information about whole disk drives, getting and setting disk volume labels, and getting and setting the write-verify switch.

Calls in the second group take as their main input a pathname in the form of an ASCII string. With these calls a program can create and delete directories, or search in them, or query or set the current directory path, or delete files, change their attributes, or move them from one path to another.

Calls in the third group take a file "handle" as their controlling parameter. They offer the ability to read or write a file, either sequentially or at random, or to change its size or its directory information, or lock some part of it for exclusive use.

Before discussing the file system calls, we will discuss the kinds of parameters they take: path strings, and words of date, time, and attribute information.

8.1 Volumes, Directories and Paths

The OS/2 disk organization is identical to that of MS-DOS version 3.3. Its structure of disk directories and subdirectories is identical to the familiar arrangement under MS-DOS, and so are its conventions for naming disks, directo-

ries and files with *pathnames*. However, there are added functions that make it easier for a program to deal with paths.

8.1.1 The Filename

Wherever they are located, files are named by a name of up to eight letters and an "extension" as long as three letters: *filename.ext*. However, this familiar form is due to be changed. One of the first enhancements slated for OS/2 is an expansion of the eight-dot-three filename to some longer form. How much longer the names will become is not known. Microsoft President Bill Gates has been heard to speak of filenames of "indefinite length" but 32 or 48 bytes seems more likely. At any rate, take pains to not write the assumption of a 12-byte filename into your program. Always make the length of a filename a named constant (it's FNAMELEN in the header files in the Disk Appendix to this book).

When used as a parameter of an OS/2 call, a filename may contain lowercase letters. However, OS/2 will always treat them as uppercase, and filenames returned by system calls are always in uppercase. That, too, seems a likely candidate for improvement. Programs should accept filenames in mixed case letters and leave them so. If the system treats them as all-cap, that's fine; if your program does, and a later OS/2 distinguishes letter case in filenames, your program will be out of step.

The system is not choosy about the format of a filename. If there are extra letters in the name, those after the eighth are ignored. Letters after the third in the extension are also ignored.

In some calls, a filename may contain asterisk and question-mark "wildcards." The use of these is familiar to any MS-DOS user. In this book, a filename that contains wildcards is said to be "ambiguous," while one with no wildcards is called "explicit."

8.1.2 The Current Drive

Each disk drive has a name that is a letter followed by a colon. The letters are established during system initialization as device drivers are loaded. Usually, the names "A:" and "B:" are reserved for diskette drives and "C:" is the first, or only, fixed disk.

For purposes of referring to files, one disk drive is the *current* drive, also called the *default* drive. This drive will be assumed unless a specific drive name is given as part of the name of a file. Each process has its own current drive, initially inherited from its parent process.

8.1.3 The Root Directory

The medium of storage in a disk drive is a disk *volume*. A diskette is a volume, as the fixed disk mounted inside a fixed disk drive. Each volume contains a *root* directory, a table of files that is always written at a fixed location on the disk surface.

In pathnames, the name of the root directory is "\"; that is, "a:\" means "the root directory of the volume in drive A:," while a pathname *\name* means, the file *name* in the root directory of the current drive.

8.1.4 Subdirectories

Other directories are essentially small files, each containing a table of files (and other directories). However, directories are restricted by special rules. Directories may only be created by *DosMkDir* and removed by *DosRmDir*, while files are created by *DosOpen* and removed by *DosDelete*. And directories may not be read and written like files, but only searched with special calls such as *DosFindFirst*.

As part of a path, a directory is referred to as *name*\; that is, "*\fred\ethel\lucy.doc*" means, look in the root directory (of the current drive) for the directory "fred," in it find the directory named "ethel," in it look for the file named "lucy.doc."

Every directory other than the root contains two special entries, "dot" and "doubledot" (. and ..). These special entries are created along with the directory and cannot be deleted from it. "Dot" names the directory itself. "Doubledot" names the parent directory, the one in which this directory is named.

8.1.5 The Current Directory Path

There is a *current* directory or *default* directory for each disk drive (not merely for the current drive). The pathstring that describes it, followed by a backslash, is assumed to precede any pathname that does not itself begin with a backslash. For

example, if the current drive is C: and the current directory is "\fred\ethel," then a reference to "lucy.doc" is actually a reference to "c:\fred\ethel\lucy.doc."

On the other hand, a reference to "\lucy.doc" (note the leading backslash) will not be qualified by the current directory; it is a reference to "c:\lucy.doc," that is, the file in the root directory of the current drive.

Like the current drive, the current directory for each drive is unique to each process. A process inherits its set of current directories from its parent process.

8.1.6 Pathnames as Parameters

Many file system calls take pathnames as parameters. Pathnames are always given in the "ASCIIZ" form, as strings of ASCII characters terminated by a byte of 00h, the null byte. This is the native way of representing strings in the C language. In other languages strings are stored in other ways, usually with a separate length field and no special terminating byte. In these languages, care must be taken to allow room for the null byte when declaring strings and to append the null byte when assigning strings.

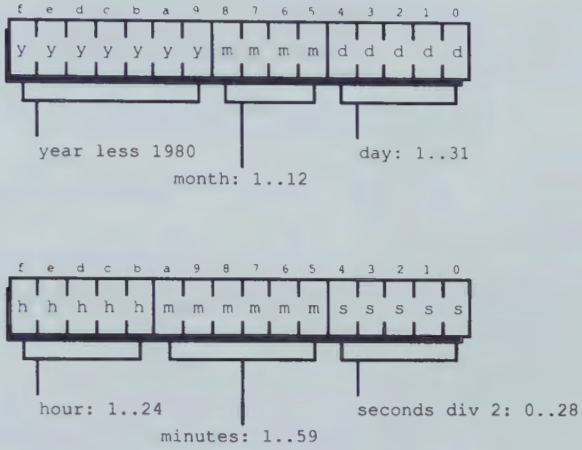
8.2 Date Words and Time Words

The system stores dates and times in disk directory entries, and these timestamps are parameters of several calls. A date or a time is stored as a 16-bit word. The day, month and year numbers of a date are packed into the 16 bits of one word, as are the hour, minute, and second of a time. The bit-level format is shown in Figure 8.1.

Seven bits are allowed for storing the difference between the year and 1980, so the possible year numbers are 1980 through 2107 a.d. In a time word, only five bits are available for encoding seconds, so seconds can only be recorded to the nearest two-second interval. The number of seconds is divided by two when encoded. When a time is decoded, the seconds number is multiplied by two to produce an even number between zero and 58.

Declarations for date and time words as data types are in the include files in the Disk Appendix. Functions for extracting the numbers from date and time words, and for constructing date and time words from numbers, are also defined there. Those for C are defined as macro expressions. For Pascal the functions are external and have to be linked from an object library.

Figure 8.1
Bit-Level Encoding of Date and Time Words



8.3 The Attributes of a File

A file's directory entry contains a set of 16 "attribute" bits; this word is a parameter of several calls. Its layout of an attribute word is shown in Figure 8.2.

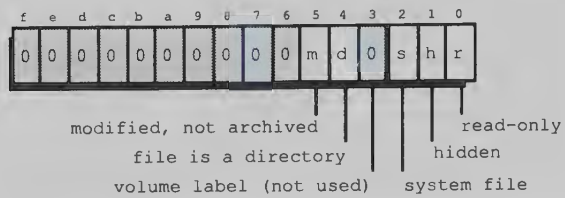
The meanings of the attribute bits are the same as in MS-DOS:

- A read-only file may not be erased, renamed, or written to.
- Hidden files and system files are both invisible to normal directory searches.
- The directory bit distinguishes a directory from a file.
- The "archive" bit is set to 1 when a file is created or modified.

The bit usually called the "archive" bit really has the true sense of "modified" or "needs archiving." It is set to 1 when the file is created and whenever it is closed after writing. It is set to zero by the BACKUP command and similar programs that archive files. (The *DosSetFileMode* call is used to change attributes.)

One bit visible in MS-DOS is not visible here: the volume-label bit. Volume labels are not returned by the OS/2 system calls that scan directories, so the bit will not appear (except in a dump of a directory).

Figure 8.2
Format of a File Attribute Word



Declarations for the attribute word as a data type are in the include files in the Disk Appendix. The Pascal files declare it as a set with five members; Microsoft Pascal maps the a small set to the bits of a word in the appropriate way.

Reference: *DosSetFileMode* 8.6.4

8.4 File Handles

A file "handle" is an index into an invisible table maintained for each process by OS/2. Whenever a file is opened, OS/2 records the fact in the table and returns the numeric index of that file's entry as the file's "handle." Each process has its own table of open files, and a process can't open more files than there are table entries. The table can be made larger with *DosSetMaxFH*.

When a process is created, it is given a table having room for twenty files. The first three, whose handles are 0, 1 and 2, are set up to describe the standard input, output, and error files respectively. These files are opened when a process is created (though possibly to the null device).

The MS-DOS system has a similar scheme of file handles. In it, handle 3 is also reserved and always open to the first printer, LPT1. That is not the case in OS/2. In order to write to the printer a process must explicitly open a file with an appropriate name, such as "PRN" or "LPT1." Any handle, not necessarily 3, might be returned for this open.

OS/2 seems to use up additional handles in the original file table, but their use is not documented. They are remnants of work done by the program loader. Their number and meaning will probably change from version to version. At any rate,

do not expect that the first available handle will be number 3; it might be any number from 3 to 20.

In sum, a process may open up to at least fifteen files before running out of handles (which causes *DosOpen* to return error code 4). The *DosSetMaxFH* call sets the size of the table larger, permitting the process to have more files open simultaneously.

The *DosOpen* call associates a file with a handle. However, the file is not the same as the handle. Somewhere beyond the table of handles is a structure, private to OS/2, that represents the files themselves, and the handle is a number that is associated with the file information.

The association between a handle and a file can be duplicated. The *DosDupHandle* call creates a second handle that is associated with the identical file structure. The handles are said to be "tied" to the same file. The same duplication takes place when a child process inherits its parent's open handles: the child's handles are duplicates of the parent's, connected to the identical file information. When handles are tied to the same file, a change in the state of the file that is effected from one handle is visible from all other handles. If the file pointer is changed from one handle, the change is effective for all handles.

However, there is a "mode" that is unique to each handle. For instance, I/O errors are reported back to a process or not, depending on the return-errors mode of the handle. This information is set by *DosOpen* and is kept on a per-handle basis. This relationship is sketched in Figure 8.3.

References: *DosDupHandle* 8.10.3: use of duplicate handles 6.1.4.3, 6.1.5.2

8.5 File System Operations

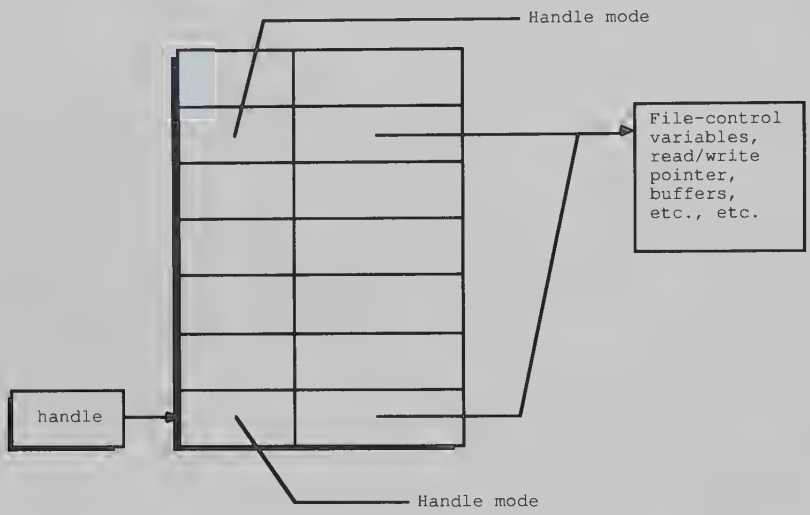
This miscellaneous group of function gives a program information about whole disk volumes, and some control over the file system.

8.5.1 Managing The Write-Verify Switch

The calls *DosQVerify* and *DosSetVerify* let a program manage the write-verify switch. When that switch is on, all disk writes are verified by reading back before they are declared complete. When it is off, disk writes are not verified. The

write-verify switch affects all disk writes done from that process, but not the writes of any other process.

Figure 8.3
Relationship of Handles to the Handle Table and to Files



Setting the write-verify switch on will at least double the time required to write disk output from the calling process, while disk output is usually quite reliable, so it isn't ordinarily done. Whenever the output, if lost to a read error, could be recovered by simply running the program again, there's little point in verifying writes.

However, there are applications that offer an implied guarantee of reliable output (a file archive program or a database system, for instance). And there are times when the data could only be recovered with significant human effort, or couldn't be recovered at all, if it were lost. In these cases, setting the write-verify switch lets the program be sure that the data have been written readably (at least, in this disk drive on this day).

Note that the program won't be notified of a read-back error unless the file was opened with the return-errors option of *DosOpen*. An error on read-back, like any other I/O error, will usually be shown to the operator in a pop-up display, and the operator will have to decide whether to retry, or cancel the program, or return the

error. A program that sets the write-verify switch is in a sense predicting an error and should ask for errors to be returned when the file is opened.

Reference: *DosOpen* mode word 8.7.3

8.5.2 Managing The Current Drive

Two calls, *DosQCurDisk* and *DosSelectDisk*, let a program set the disk drive that will be the current drive, used by default in file and directory operations. The current drive, like the write-verify switch, is peculiar to a process; different processes may have different current drives. For that reason, if one process passes a pathname to another process (through shared storage or through a file), it should pass an explicit drive letter as part of the path-string, or arrange to pass a drive number separately.

The *DosQCurDisk* call also returns a drive "map," a string of 32 bits containing a 1-bit for each drive letter that exists in the system. This can be used to validate drive letters taken from operator input.

8.5.3 Managing The Current Directory

Two calls, *DosQCurDir* and *DosChDir*, let a program query and set the current directory path for a drive. The query call returns the directory path string for a specified drive. The returned string doesn't contain a drive-letter or a leading or a trailing backslash. If the aim of getting it is to put it together with a filename to make a fully-qualified pathname, it will need this punctuation added to it. The steps to forming a fully-qualified pathname are:

- Specify the drive as "X:" using the current drive's letter.
- Append a backslash to start the path at the root.
- If the current directory isn't the null string, append it and another backslash.
- Append the "filename.ext" of the file.
- Append the null terminator if that isn't automatic in the programming language in use.

The *DosChDir* call sets the current directory path for any drive. Its path parameter may contain a drive letter, which causes the given path to be set for that drive, not the current one. Like any pathname, it may start from the root or it may build on the existing current directory.

The longest string that *DosChDir* will accept is 64 bytes. What's more, it won't accept a shorter string that lacks a leading backslash, if it and the current directory path would sum to exceed 64 bytes. It's possible to build a directory structure so deep that a fully-qualified path through its furthest directory exceeds 64 bytes—but it's not a good idea to do so. It won't be possible to make that furthest directory the current directory or to refer to it with a full pathname.

This sets an upper limit on the space a program needs to reserve for pathnames. The longest directory prefix is 64 bytes. Add two bytes for an "X:" drive letter, two for extra backslashes, twelve for the "filename.ext," and one for the null terminator. The longest possible path string is then 81 bytes. But beware! A future system that allows longer filenames, and thus longer directory names, will surely permit longer path strings. Use a named constant for the limit (it's FPATHLEN in the Disk Appendix).

8.5.4 Querying File System Info

The *DosQFSInfo* call is really two calls, depending on a "function" number parameter. One function returns the important numbers about a disk drive:

- how big a "sector" is, in bytes,
- how big an allocation unit is, in sectors,
- how many allocation units the disk contains, and
- how many of them are free.

A "sector" need not be identical to a physical disk sector, but it will represent the smallest unit of data that is read or written from the disk and hence, the smallest efficient buffer size for reading and writing a disk file.

The size of an allocation unit is the smallest amount of data space that can be allocated to a file; a file smaller than one unit will nevertheless occupy a full unit on disk. The total number of units tells the largest possible single file that this disk volume could hold.

The number of free units suggests, but does not guarantee, how much more data a program could write to this volume. There are other programs in the system, and even as this program finds out how much space there is, another program could be filling it up or deleting a file.

This call also returns a four-byte "file system ID." At present this is always binary zero. In future systems there may be more than one "file system" in opera-

tion. Then a specific drive might be controlled by a different package of code than the usual file system (by the LAN manager or a CD-ROM driver, for instance). For this reason an OS/2 program should never make assumptions about how files and directories are laid out on disk, or should at least check this field to make sure that the old, MS-DOS type of file system is managing the drive.

8.5.5 Managing The Volume Label

The second function of *DosQFSInfo* returns the volume label from a drive, including its date and time of creation. A matching call, *DosSetFSInfo*, will write a new volume label. Unfortunately for consistent programming, this call does not take the same data structure for input that the query call returns as output.

8.6 Directory Operations

This group of functions gives a program control over the structure of directories on a disk. They make it possible to make and remove directories and to move and delete the files named in directories. Other calls make it possible to retrieve the information kept in directory entries, and to update some of it.

8.6.1 Making and Removing Directories

The *DosMkDir* and *DosRmDir* calls operate very much like the "md" and "rd" commands. The first call creates a new directory that contains only the "dot" and "doubledot" entries. The second call will delete a directory, provided it is empty of all entries except the dots.

It is interesting to note that *DosMkDir* does not reject path strings longer than 64 bytes, as *DosChDir* does. This makes it possible for a program to create a directory structure such that the innermost directories can never be the current directory. Fortunately, *DosRmDir* is able to remove anything that *DosMkDir* can make.

8.6.2 Renaming and Moving Files

To *rename* a file is to change its filename. This is done by updating the directory entry that describes the file. To *move* a file is to change the directory in

which it is described. This is done by deleting its entry in one directory and adding it to another directory.

Both of these functions are performed via the *DosMove* call. It takes two path strings, an old one and a new one. The old one must describe an explicit file that exists, while the new one must describe a path of directories that exists and a filename that doesn't.

If the path strings differ only in the filenames that end them, *DosMove* performs a rename, changing the filename in the present directory. However, if the paths differ (whether or not the filenames do), a move is implied. The entry that describes the existing file is moved to a different directory.

The file's contents don't move in any of these operations; only the two directories are altered. There is no copy of file data, only the pointers to data are copied. That's why *DosMove* won't accept paths that point to different disk drives: since it doesn't actually copy the file, it would make no sense to move its directory entry to a different drive.

The *DosMove* operation preserves file attributes, dates, and times. It works on all files, including those with the read-only, hidden and system attributes. It does not work on directories, and it returns an error code if asked to move a file that is in use by another process.

8.6.3 Deleting Files and Directories

Files are deleted by pathname. The *DosDelete* call takes a path string and deletes the file it describes. The path must name a file explicitly; wildcard characters are not allowed. In this, OS/2 is more restrictive than earlier systems which allowed multiple files to be deleted in one call.

Directories are a special case in this as in other things; they are deleted using *DosRmDir*, not *DosDelete*. *DosDelete* is used to clear out all the files within a directory so that *DosRmDir* will accept it.

There are two reasons why an attempt to delete a file may fail. The file may be read-only, or it may be in use by another process. These produce different error codes and they ought to be reported to the user with different messages. The user must take different actions to deal with each case.

8.6.4 Setting File Attributes

The attribute word of a file can be retrieved from its directory entry by a call to *DosQFileMode*. It takes a path string that explicitly names a file or directory, returns the attributes of that file or directory. This lets a program check to see if a particular file is read-only before trying to erase or rename it, or to see if it has been modified since archiving, or to see if it is marked hidden or system and hence ought not to be listed for the user.

A file's attribute word can be updated via *DosSetFileMode*. The read-only, hidden, system and archive attributes can be altered. The attributes of a directory cannot be changed however, nor can an ordinary file be made a directory.

The *DosSetFileMode* call is designed to be used in tandem with *DosQFileMode*. Apply the query call to get a file's attribute word, change in it only the bits of interest, and update it.

8.6.5 Scanning a Directory

Four system calls work together to let a program scan the contents of a directory, getting the names and detailed characteristics of the files in it. Similar functions existed in MS-DOS (and in CP/M before it), but the OS/2 functions are more sophisticated. One, *DosSearchPath*, makes it easy to find a file anywhere in a list of paths (such as the list of paths stored in the PATH and DPATH environment variables). The others, beginning with *DosFindFirst*, set up a directory scan.

8.6.5.1 Searching a Path

The familiar shell command, Path, sets a list of path strings in the environment. (The command "SET PATH=list" has the same effect.) The protect-mode command shell also supports a Dpath command and matching environment variable. A path list is just some path strings separated from each other by semicolons. It's a simple idea and one that users have been trained to use, so it would be nice if other programs could use similar path lists. For instance, it would be handy to be able to tell the user to "set the SEARCH environment variable to the list of paths the program should search," or to "type a list of paths similar to that used in the PATH command."

Unfortunately the code to parse a list of paths, extract the paths one by one, and test each for the presence of a file is tedious to write. It's good news that the *DosSearchPath* does it all. It takes two main parameters, a list of paths and a filename, both in ASCIIZ form. It extracts each path in turn, appends the filename to it, and searches for a matching file on that path. At the first match it stops and returns the complete file path as an ASCIIZ string.

The subtleties of path processing can cause difficulties for the program or its users. Each member of the path list is extracted and prefixed to the sought filename. If the path begins with a drive letter, that drive will be used; if not, the current drive will be used. This can baffle users ("But *why* doesn't it work after I move to the floppy?"). If a path applies to only a certain disk then be sure to include the disk drive letter in the path.

In the same way, if a path list member doesn't have the initial backslash that starts it at the root, it will be prefixed by the current directory path. This too, can cause puzzlement. Most members of a path list should probably start with a drive letter, colon and backslash.

DosSearchPath takes two options. One tells it whether or not to search the current directory before beginning to search the paths in the list. It's a program design choice whether the current directory has priority. Note, however, that the current directory can be inserted in the path list very easily. A null list member—two adjacent semicolons—is a reference to the current directory on the current drive. So is a single dot, or a drive letter and colon alone or with a dot. These make it easy to put the current directory at the head, tail, or any midpoint of a path list, but take pains to point that out in the program documentation.

The second option of *DosSearchPath* tells it whether the path list has been passed directly, or indirectly via the environment. If the list comes directly from a program variable, it would of course be passed directly. In the indirect case, the parameter of the call isn't a path list at all, but the *name of an environment variable*. For instance, to look for a file anywhere on the shell's execution path, the program need only call *DosSearchPath* with the string "PATH" and the name of the file. OS/2 will look up the name in the current environment and use its value as the path list to search.

DosSearchPath doesn't require that the filename be explicit; it may have wildcards. The wildcards are preserved in the output string. This makes it possible, for instance, to find the first occurrence of "Z*.DOC" along the DPATH, and then to use the *DosFind* calls to get the names of all such files.

8.6.5.2 The DosFind Calls

The *DosFind* calls return the contents of a directory one file at a time. *DosFind-First* begins the scan. It takes a path string and returns information about the first file that matches. It also returns a handle that stands for the scan for files that match that string. The search may be continued with *DosFindNext*; it takes a scan handle and returns the next files that match. The handle is given up by calling *DosFindClose*.

These search functions are restricted in one way compared to MS-DOS; they don't return volume labels. The volume label is queried or set with the *DosQFSInfo* and *DosSetFSInfo* calls.

The use of a handle to stand for a particular search is an advance on older systems; it means that a program can have several independent directory scans going on simultaneously. (In fact, a program could have different scans going on concurrently in different threads.) Each scan is separate and is not affected by other scans or any other file operations.

Another advance is that these calls are able to return data about more than one file at a time. In fact, information on a dozen or a hundred files can be retrieved in one call, if a large enough buffer space is provided for output.

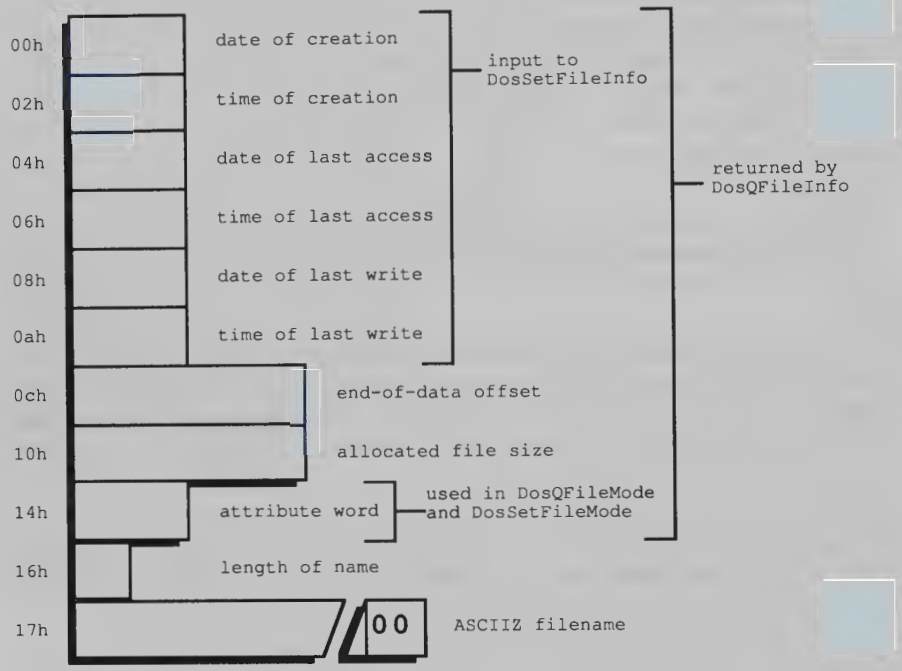
8.6.5.3 The DosFind Data Structure

The data structure returned by the *DosFind* calls is quite complex (see Figure 8.4), and not all of it is meaningful in the present system.

The date and time fields in this structure have the form discussed in Section 8.2, and the attribute word has the form discussed in Section 8.3. The attribute word can be altered using *DosSetFileMode*. The date and time fields can be updated with *DosSetFileInfo*.

In version 1 of OS/2, the creation date and time and the last-access date and time are not maintained; only the date and time of last write are actually kept in the directory.

Figure 8.4
Data Structure Returned by DosFind and Other Calls



It isn't known when the system will start maintaining all three timestamps. Even when it does, there will always be mountable volumes whose directories were written at a time when it didn't. Therefore a program should be written so that it will use these items if they are present, and not if they aren't. For most files, the time of last write is also the time of creation, since most files are written only once, when they are created. Therefore if the creation date is zero, use the date and time of last write in place of the creation date and time.

The time of last access, that is, last read or write, simply can't be known. It cannot be earlier than the last write, so for some applications the time of last write may be an adequate substitute.

The End-of-data offset and the file size in Figure 8.4 are both 32-bit integers, since a file may be as large as 32 megabytes, two to the 25th power, and later versions of the system are expected to support larger files still.

The end-of-data offset is also the count of data bytes in the file. This is the number that would be reported as the exact size of the file, and the number that would be the basis for calculating the allocation size of the file on a different disk format. It is the offset to which the file pointer would be set (with *DosChgFilePtr*) in order to append data to the file.

The allocation size is the amount of disk space that would be freed if the file were erased. In an ordinary file created by sequential output it will be the end-of-data offset, rounded up to a multiple of the size of one allocation unit on this disk (allocation unit size is returned by *DosQFSInfo*). However, it is possible to give a file a larger size with *DosOpen* or *DosNewSize*, so the allocation size might be much larger than the data in the file.

The name field at the end of the structure contains the explicit filename as an ASCIIZ string. This field has a variable length. The length byte at offset 16h gives the length of the actual name, exclusive of the null terminator. (Thanks to the length byte, the field beginning at offset 16h may be declared in Microsoft Pascal as an *lstring*.)

In the present OS/2, the longest possible filename is twelve bytes; that plus the null makes the longest instance of this data structure 35 bytes. However, Microsoft has promised that a future system will allow longer filenames, so it is a good idea to allow some slack in this data structure, so that if the next version of the system supports longer filenames, the program might still work without recompiling. The include files in the Disk Appendix declare the string itself as 41 bytes, making the structure a nice round 64 bytes long.

8.6.5.4 Using DosFind With Other Calls

The calls to delete, move and change attributes require explicit filenames. The *DosFind* calls allow a program to find all the files that match an ambiguous filename, which means they afford a way to apply the other calls to multiple files—to make "*.asm" read-only, for instance, or to delete "x??z.doc."

In the simplest case the path string has only a filename, no preceding directory names. If the program is sure of that, it can proceed this way:

1. Call *DosFindFirst* with the path and a count of 1. If it returns a count of zero, there are no matching files; report the error and stop.

2. There is at least one matching file and the explicit filename is in the data structure returned by *DosFindFirst* or *DosFindNext*. Use the filename as the input path string to *DosMove*, *DosSetFileMode*, or *DosDelete*.

3. Call *DosFindNext* with a count of 1. If it returns a count of 0 there are no more files, stop. Otherwise there's another file, repeat from step 2.

Things get more complicated when the path string is allowed to contain directories and drive letters. The problem is that the *DosFind* calls return only the explicit filename; they don't return the full pathname of the found file. Yet the full path string is needed as input to the other calls such as *DosDelete*.

A partial solution is to extract from the path string a *prefix* path: the drive and directory names that should qualify any explicit filename. The simplest way to do this is to scan the path string from right to left looking for the rightmost ":" or "\" byte. If there is none, the prefix string is null; otherwise it is all bytes up to and through that byte.

Set up a copy of the prefix substring in a separate variable. Then apply the general algorithm listed above, but in step 2, form the input path string by copying the prefix string and appending to it the explicit filename returned by *DosFind*.

The Disk Appendix contains some simple utilities that illustrate this method, and some minor enhancements of it.

8.6.5.5 Handling Multiple Found Files

It is not difficult to process the data returned by *DosFind* when only one instance is returned. Simply pass to the *DosFind* call the address of a variable of the appropriate size and an input count of one, and process the single data structure it returns.

When a *DosFind* call is asked to return more than one structure, it packs them as tight as possible in the output buffer—the second follows immediately on the null terminator of the first. This means that a multiple response can't be declared as an array of structures. Instead, the program has to set up a pointer variable to the first structure, and then increment that pointer by the exact length of the current structure in order to locate the next structure.

This is an awkward thing to do in any language. In Pascal, it can be done using the nonstandard ADS type. In the Disk Appendix, the structure of Figure 8.4 is

declared as a FileFind record. A pointer to FileFind and a buffer area might be declared as

```
 Var FF   : ADS of FileFind;
ManyFFs   : array[1..MaxFF] of FileFind;
```

The pointer can be set to the address of the buffer,

```
FF := ADS ManyFFs[1];
```

and then "FF@" may be passed as the buffer parameter of the *DosFind* call. "FF@" is also the name of the first returned structure, and the next one is located by a statement such as

```
FF.R := FF.R + 16#18 + FF@.fname[0];
```

The C include file in the Disk Appendix declares the structure as *struct FileFind*, and declares the *DosFind* calls as taking a parameter of *struct FileFind far** for the buffer. A pointer variable and buffer space might be declared as

```
struct FileFind manyff[MAXFF];
struct FileFind far *ff;
```

Then *(ff=manyff)* might be passed as the parameter to the *DosFind* call. The first returned structure is now available using *ff->*; however a simple increment of such a pointer will increment it in multiples of the value of *sizeof(struct FileFind)*, which won't work. The pointer must be incremented using casts, something like this,

```
ff = (struct FileFind far *)(0x18 + ff->name_len+(char far*)ff);
```

casting the structure pointer to a character pointer and casting the result back to a structure pointer. On the whole it would have been nicer if the OS/2 designers had been less careful of saving every byte, but it's too late now.

8.6.6 Getting and Setting File Dates

Two functions make it possible to retrieve and update the file timestamp information. These functions are logical cousins to *DosQFileMode* and *DosSetFileMode*, which take path strings as parameters. Strangely, the *DosQFileInfo* and *DosSetFileInfo* calls do not take path strings to specify a file.

Instead they require a file to have been opened with *DosOpen*, and then take the file handle returned by *DosOpen* to name the file.

Suppose a file has been opened by *DosOpen*. Then *DosQFileInfo* on that handle returns a portion of the data that would be returned for that file by a *DosFind* call (see Figure 8.4). The returned data contains all the timestamps, the end-of-data and allocation sizes, and the attribute word. As discussed 8.6.5.3, some of the timestamps aren't maintained in version 1 of OS/2.

The *DosSetFileInfo* call takes an even smaller portion of the information structure (see Figure 8.4), but it updates that portion of the directory entry. It is possible to set new dates in a file this way.

DosSetFileInfo has been designed to work in tandem with *DosQFileInfo* (or with the *DosFind* calls). Retrieve the file's dates with one of the query calls, alter only the fields of interest, and update the fields with the set call.

8.7 The DosOpen Call

The *DosOpen* call has many functions and a bewildering number of parameters. Most programmers will find it convenient to build simpler subroutines that take only a few parameters and call *DosOpen* for a specific operation. Here we examine its parameters first, then consider how to use them in specific cases.

8.7.1 The Open Path String

The first parameter to *DosOpen* is a path string. As with any path, the current drive will be assumed unless it starts with a drive letter, and the current directory path will be prefixed to it unless it specifies the root directory. Ordinarily the path will name an existing disk file or a disk file to be created, but it may also name one of the standard devices, and in the special "DASDOpen" case it may have just a drive letter and colon.

8.7.1.1 Opening Device Names

The device names that are supported are:

- NUL for the null device
- PRN or LPT*n* for a printer

- COM*n* for a serial port
- CON for the console
- KBD$ for the OS/2 keyboard driver
- SCREEN$ for the OS/2 screen driver
- CLOCK$ for the clock driver
- MOUSE$ for the mouse driver

The NUL device may be used for testing and as a destination for unwanted output. An input file opened to "NUL" will see immediate end of file; output to a file opened to "NUL" is discarded.

The PRN and LPT*n* devices represent printers. The use of the printers is not serialized (even by the share options discussed below); any number of concurrent processes may open a printer and write to it. Their output will be mixed on the page unless the SPOOL command has been used to start the print spool subsystem. It segregates printer output into separate streams by process.

The CON device represents the logical screen and keyboard. Files opened to "CON" are *not* necessarily the same as the files represented by the standard handles 0, 1 and 2. Reading from a file opened to CON always reads from the keyboard (if one exists) while reading from standard input (handle 0) may read from the keyboard or from another source, depending on whether or not standard input was redirected by the parent of this process. Similarly, writing to a file opened to "CON" always goes to the screen (if one exists), whereas writing to standard output or standard error might go to a redirected destination.

When a process is running detached (started by the Detach command) it does not have access to a keyboard and screen. In that case, files opened to "CON" are effectively opened to "NUL."

The "KBD$" and "SCREEN$" devices underlie the "CON" device. If it is necessary to control either device directly, it is best to use the KBD- or VIO- system calls discussed in other sections. There is one use for opening a file to "KBD$," however: its handle may be interrogated to see if the process is detached.

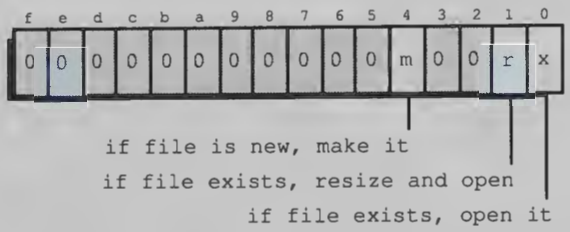
References: discovering if detached 11.1; keyboard functions ch. 11; display output ch. 10

8.7.2 The Open Flag Word

The "Flag" parameter specifies what is to be done if the specified file exists, and

what if it does not. The answers to these two questions are encoded in three bits of a word, as shown in Figure 8.5.

Figure 8.5
Format of an Open Flag Word



When the specified file doesn't exist there's only one choice: should it be created or not? The answer depends on what the program wants to do. If it expects the file to exist—if it expects to read from it, or if the file is a device—then the answer is no, don't create it, return an error. But if the program means to write a new file, then creating a new file is just the ticket. This answer is coded in a single bit of the Flag word.

When a file is created during open, its size and file attribute word are set to values passed as parameters to *DosOpen*.

There are three ways that *DosOpen* can handle an existing file. It can go ahead and open the file—the obvious right choice when the program expects the file to exist. Or *DosOpen* can stop and return an error code. This is the choice when the program does not expect or want the file to exist.

Finally, an existing file can be resized to the size given in another parameter, and opened. This rather unusual choice is right when (for example) the file is to be rewritten from scratch. (The OS/2 manuals may refer to this as the "trunc" option, but that's a misnomer. The file can be truncated to a smaller size, or it can be inflated to a larger one.)

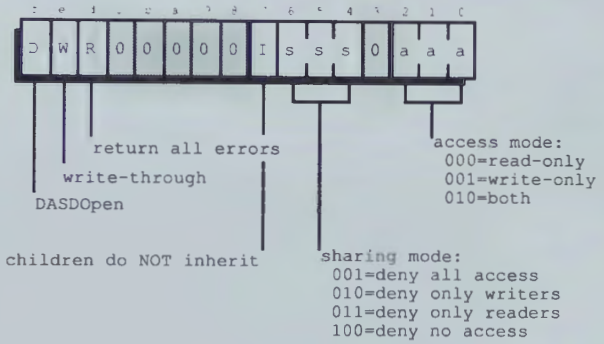
One of the *DosOpen* parameters is a variable, the "action" word. This returned value is only relevant when the Flag word specifies both making a new file and opening (or resizing) an existing one. Then the returned "action" value lets the program know whether the file existed or was created.

8.7.3 The Open Mode Word

The "Mode" parameter specifies a number of options. It is a set of bits formatted as shown in Figure 8.6. The mode is stored with the file handle and qualifies operations through the handle (see Figure 8.3). It is specified when the file is opened, but the same bitset can be queried, and even changed, later while the file is being used.

Most of the contents of the Mode word are discussed in this chapter. One, the DASDOpen bit, is discussed in chapter 15.

Figure 8.6
Format of an Open Mode Word



8.7.3.1 The Access Options

The least significant bits of the Mode word specify the program's intentions toward the file: whether it will only read it, only write it, or might do both. These intentions are important when files are shared by multiple programs (as any file might be). Some files can be shared only by readers, while others can be shared with programs that intend to write. Don't automatically specify "both" to be on the safe side; specify exactly and only what the program plans to do.

8.7.3.2 The Share Options

Bits 4, 5, and 6 of the Mode word specify the program's willingness to share the file with other, concurrent programs. The options are specified negatively, in terms of what kind of concurrent uses this program must deny to others.

The most generous option is to "deny none," that is, to let the same file be used by any other process including one that wants to write to it. A file that's open "deny none" still can't be deleted, renamed, or moved—but it can be opened with the "resize" flag option and resized to zero, or written over.

The "deny writers" option lets the file be used by any number of other concurrent processes, so long as they open it with the "only read" access mode. This is a good sharing option for any input file.

The "deny readers" option shares the file only with programs that open it with the "only write" access mode—a strange choice, but someone will find a use for it.

The "deny all" option insists on exclusive use of the file; no other process may open it while this program has it open. Use it only when there's a good reason for it; if the program only reads a file there's no reason to deny access to other readers.

It is often important to have exclusive use of a device. Unfortunately, the Share option is ignored when the file is a device; opening a device with "deny all" is the same as opening it "deny none." Whether concurrent opens are allowed is determined by the device driver that implements the device.

8.7.3.3 The Inherit Mode Option

Bit 7 of the Mode word indicates whether a child process should inherit this file. It's not relevant unless this process will start another one with *DosExecPgm*.

If this Inheritance option bit is specified as zero, a child process will be able to use the same handle number to access the same file without need to open it. If the Inherit bit is 1, the handle won't be active for the child process—it will be an available handle that might or might not be used when the child process does a *DosOpen*.

Reference: *DosExecPgm* 6.1.2

8.7.3.4 The Return-Errors Option

A number of serious errors can occur during file access, for example errors reported by the hardware. Ordinarily OS/2 reports these directly to the operator in a pop-up menu. The operator gets to choose between ignoring the error or retrying it or terminating the program—or returning the error to the program.

Bit 13 of the Mode word can be set to 1 when the file is opened (or later via *DosSetFHandState*). This requests that all errors be reported back to the program at once, without notifying the operator. Consider carefully before specifying this option. True, it gives the program access to errors such as 21, device not ready, and 28, printer out of paper, that the program might want to handle in its own way. However, it also returns errors like 23, data error on read, that the program hasn't any good way of handling.

8.7.3.5 The Write-Through Option

The disk-caching methods of OS/2 aren't public knowledge, and they may change from version to version. Still, it's safe to say that ordinarily OS/2 may defer disk writes for some time after the program has called *DosWrite*. It might save up data until it has a large physical unit to write, an entire sector or track. Or it might queue the disk output requests of all processes in track sequence to optimize the disk arm motion.

Ordinarily this doesn't make any difference to the program. The system takes the data; eventually it shows up on disk; and nobody cares how long it takes or whether it's written in the same sequence the program used it. But sometimes it is important to be absolutely certain that data has been written to disk before going any further in the program. When that requirement applies, bit 14 of the the Mode word can be set to 1. Then whenever the program calls *DosWrite*, control doesn't return until the data has reached the disk.

If only certain passages of output are critical, consider the *DosBufReset* call as a way of ensuring file integrity.

References: *DosWrite* 8.11.3; *DosBufReset* 8.11.6

8.7.4 The Size and Attribute Parameters

DosOpen takes a file-size parameter. This parameter has effect only when a new file is created, or an existing file is resized, during the open. Note that the resizing affects both the file's allocated size on disk and its end-of-data size (both values are returned by *DosQFileInfo*). The end-of-data value is set to the exact size parameter value, while the allocation size is set to the next higher allocation boundary.

DosOpen also takes an attribute word parameter (see Figure 8.2). This parameter has effect only when a new file is created during open (however, it is checked for validity during *any* open, whether it is used or not). It specifies the attributes that the new file will have. Be careful in setting the hidden and system attributes, as they make it impossible to name the file in any commands, including Copy and Attrib.

8.7.5 File-Open Scenarios

Let's consider some common ways of using the very flexible *DosOpen* call.

Probably the most common use of a file is sequential input. In this case the file should exist, and the *DosOpen* parameters (in the order they're written) are:

- the path to the desired file
- the handle
- a scratch word for discarding the "action" number of 1
- an irrelevant size value of zero
- an irrelevant file attribute of zero
- a Flag of open-existing and fail-if-new
- a Mode showing access of read, share of deny-writers, and possibly Inherit

The next commonest use is to open a new file for sequential output when no file of the same name is expected to exist. The *DosOpen* parameters would be:

- the path to the file
- the handle
- a scratch word for discarding the "action" of 2
- the new file's size, usually zero
- the new file's attribute word, usually zero
- a Flag of make-if-new and fail-existing
- a Mode for access of write, share of deny-all, and possibly Inherit

If this call returns error code 110, a file of this name does exist. What the program does in that case depends on the application. If an existing file cannot be important, the Flag above should request resizing an existing file. Then the question can't arise, and the action word would show whether the file had been created or resized. If an existing file might be important, the program will have to take the error return as a signal to do something—perhaps use *DosMove* to rename the existing file to .BAK, before repeating the *DosOpen*.

Reference: *DosMove* 8.6.2

8.8 Closing a File

Compared to opening one, closing a file is child's play. The *DosClose* call takes any handle produced by *DosOpen* and severs the connection between the file and that handle number in that process. This is a bookkeeping process, but depending on the actual file, other things may be done. For disk output files, buffered output is written and the disk directory is updated. (A similar effect can be had at any time by calling *DosBufReset*.)

For device files, the device driver is notified of a close operation, a notification that most device drivers ignore. However, the print device driver passes the notification along to any monitors that may be registered to it. The SPOOL command is one such. If it has been diverting a stream of print data to disk for this process, it will see the close and will close its disk file.

The connection between the file and any other handle number, such as might have been produced with *DosDupHandle*, is not affected. Nor is there any effect on the association between the file and a handle in a child process that might have inherited the handle.

8.9 Changing a File's Size

The size of an input file is what it is, and it's adequate to let a sequential output file grow to whatever size it needs to be. However, a file used for direct access may need to have a certain size. if you know that a file will eventually be hundreds of kilobytes or some megabytes in size, and that it will be used for direct, not sequential, access, there is an advantage to forcing it to that size immediately rather than letting it grow by increments. The advantage is that the file's space

will be allocated as compactly as possible, while if it is allowed to grow there may be large gaps between its parts.

Here are all the ways to change a file's size. *DosOpen*, when it creates a new file, will give it the size specified. *DosOpen*, used to open an existing file, can be made to resize it either larger or smaller. A file can be extended by writing at some offset beyond its end, so one way to give a file a size *n*, is to set the read-write pointer to *n* and to write one byte. Finally, *DosNewSize* takes a handle and a size and sets the file's size either larger or smaller.

Changing a file's size is a bookkeeping operation that affects only directory and space-allocation information. The space allocated to the file will contain residual data, either disk formatting bytes or portions of deleted files. If the program requires a file to contain some default contents, the file will have to be written out to size.

Reference: file read-write pointer 8.11.5

8.10 Operations on Handles

Some operations work on handles as subjects, rather than on the files represented by the handles.

8.10.1 Getting and Setting the Mode

DosQFHandState call returns the mode word of the handle (see Figures 8.3 and 8.6), while the *DosSetFHandState* call updates some of the bits in it. The bits that may be altered are the Inheritance, Write-Through, and Return-errors options. The program may want, for instance, to turn on the Write-Through option while updating some especially critical portion of the file.

The *DosSetFHandState* call is designed to work in tandem with *DosQFHandState*. Use the query call to retrieve the handle's mode word—do *not* assume it will be identical to the mode word parameter given to *DosOpen*—alter only the bits of interest, and use the set call to update the word.

8.10.2 Testing the File's Type

The *DosQHandType* call allows the program to peep beyond the handle at the file

it names. It returns a flag that says whether the handle represents a file, a pipe, or a device. If the file is a device, the call also returns the device-driver attribute word of the driver that controls the device.

There is very little useful information to be extracted from a device driver's attribute word. However, there are four bits that can be of interest: the least significant four bits have these meanings:

- bit 0 is true for the keyboard driver
- bit 1 is true for the screen driver
- bit 2 is true for the nul device
- bit 3 is true for the clock device

A program can use *DosQHandType* to find out if a file is the null device. The call can also be applied to the standard handles 0, 1 and 2. If these are not associated with devices, or if the devices are the null device, a program may assume that its standard files have been redirected.

8.10.3 Duplicating Handles

It is sometimes useful to change the connection between handles and files. This can be done using the *DosDupHandle* call. This call takes an open handle and another handle number, and ties the second handle to the identical file as the first (see Figure 8.3).

The *DosDupHandle* call can be used in two different ways. The first way is simply to coin a new handle that refers to the same file. When it is passed a second handle of -1 (all-ones binary), the call allocates an unused handle and ties it to the same file as the first. The program now has two handles that are associated with the identical file. Either may be used for file operations or they may be used alternately, it doesn't matter. If one is closed, the other will remain open and usable.

The second way to use the call is to pass a specific second handle, which must be one of the standard numbers (0, 1 or 2) or the handle of another open file. The second handle is closed as by *DosClose*, ending its assocation with whatever it was opened to; then it is tied to the identical file as the first. This is similar to an assignment to a pointer variable; it changes the second handle so that it points to a different file.

These two functions can be applied to control the meanings of the standard files received by a child process. The specific methods are discussed in the topic on child processes.

Reference: use of *DosDupHandle* 6.1.4.3, 6.1.5.2

8.11 Input and Output

The main point of opening a file is to read or write data. The OS/2 calls to do that are simple and flexible.

When a file is opened to a disk file, one of the variables set up by OS/2 is a *read-write pointer*, a 32-bit integer that is the index of the next byte in the file to be read or written. The first byte in the file is byte number zero, and the initial value of the read-write pointer is zero. Every read or write advances the pointer by the number of bytes transferred. The pointer can be set to any value with the *DosChgFilePtr* call. That's how random access is done.

The read-write pointer is associated with an open file and *not* with a handle. If more than one handle is tied to a file, a change of the pointer through any handle changes the pointer for all handles—even if the change is done from a different process.

8.11.1 Reading from a File

The *DosRead* call takes a handle, a buffer, and the number of bytes wanted. It attempts to fill the buffer with data from the file, beginning with the byte indexed by the current read-write pointer. The number of bytes actually transferred is returned, and the file's read-write pointer is advanced by that count.

OS/2 imposes no meaning, no data typing, on the input bytes. It does not recognize ASCII line delimiters or logical record formats of any kind; all interpretations of that sort are up to the program. Input of the byte value 26 or 01ah, sometimes the logical end of file byte, gets no special recognition from *DosRead*.

If the data are supposed to be logical records of a certain size, the program will usually request input of the length of one or more records. If the file is ASCII text, the program will typically allocate a buffer of some convenient size—usually a power of two such as 512 or 1024—and use it as a stock from which to

draw bytes one at a time to form logical lines. Whenever the buffer is empty the get-next-byte subroutine will ask *DosRead* to refill it.

Most commonly the file is a disk file and not near its end, so the result of *DosRead* is to read as many bytes as were requested. When the file pointer nears the end of a disk file, some *DosRead* call will return a count smaller than the count requested. This should *not* be taken as a sign of end-of-file. Finally a call to *DosRead* will return a count of zero bytes transferred; this is the end of file signal.

When the file is a device, *DosRead* may return fewer bytes than requested. For example, a read from the console device will return no more than one line no matter how many bytes the program requests. When the file is a pipe, *DosRead* returns as many bytes as are in the pipe. In this and other cases, a short but nonzero return count indicates a shortage of data but not necessarily end of file.

8.11.2 Reading from Devices

Here are some peculiarities of reading from devices. Reading from an output-only device such as a printer causes no error; it results in a returned count of zero, indicating end of file. So does reading from the null device.

A six-byte sequence can be read from the CLOCK$ device driver that is implemented in OS/2 for the IBM PC/AT and compatibles (this is quite likely hardware-dependent information; do not assume it is correct in every system). The contents of the six bytes for the PC/AT clock hardware are

- a word, the number of days since 1-1-1980
- a byte, minutes in the hour
- a byte, the hour from 1-24
- a byte, hundredths of a second from 0-99
- a byte, seconds in the minute.

Better time information is available from the *DosGetDateTime* and *DosGetInfoSeg* calls, without hardware dependencies.

When reading the CON device, the first read causes a physical read to take place that doesn't end until the return key is pressed (similar to a call to *KbdStringIn* with the Wait option). Both a CR and an LF are appended to the line, presumably so as to make input from CON identical to input from an ASCII disk file. If the resulting line is shorter than the requested count, the whole line is returned

and its length is the returned count. Otherwise only as many bytes are returned as were requested and the remainder of the line is saved for the next read.

Input from a COM device can return a count of zero after an input timeout. The serial ports and their timeout policies are discussed in chapter 14.

The mouse device driver will accept an open, but an attempt to read from the resulting handle will produce critical error 22, unknown device command.

> **References:** *DosGetDateTime* 5.1.2; *DosGetInfoSeg* 5.1.4; *KbdStringIn* 11.5; serial port I/O ch. 14

8.11.3 Writing to a File

The *DosWrite* call takes a handle, a buffer, and a count of bytes to write. It copies as many bytes from the buffer to the file as it can—usually all that were requested. The read-write pointer is advanced by the number of bytes written.

In sequential output, bytes always go to the end of the file, making it longer. Allocation units are added to a disk file as necessary to hold the new data. However, in a disk file, the read-write pointer doesn't have to be at the end during output; it is set to the start of an existing file by *DosOpen* and can be set to any point in the file with *DosChgFilePtr*. In this case, the output bytes are written over the existing bytes.

OS/2 imposes no special interpretation on the data; it doesn't look for ASCII line delimiters or mark logical record boundaries in any way.

8.11.4 Writing to Devices

In contrast to input, output to devices has few differences from output to disks. Output to CON and SCREEN$ devices will be mixed with output from other processes active in this screen group, if there are any.

Output to a printer will come to the printer device driver mixed with the output of other processes. However, this mixed stream of output will be passed on to any monitors that have registered to the printer driver, and one of these may disentangle it. Specifically, a print spooler program may segregate the streams of print output from different processes, buffer some on disk, and send only one on to the printer.

Output to a COM device will be buffered and sent by the serial port device driver.

The mouse device driver can be opened but an attempt to write to it will produce critical error 22, unknown device command.

References: screen group 1.2.1; display output ch. 10; serial port output ch. 14

8.11.5 Asynchronous Reads and Writes

File reading and writing can be time-consuming, and sometimes the time is consumed in nothing more than waiting for a device to work—waiting for a disk to seek or waiting for a slow communications line, for example.

While your program is stuck waiting for I/O, OS/2 will try to find another program to make use of the time. You may feel that your own program could be doing productive work while the I/O operations are going on. In that case you can start a read or a write operation *asynchronously*.

The *DosReadAsync* and *DosWriteAsync* calls ultimately have exactly the same effects as *DosRead* and *DosWrite*. The only difference is that OS/2 creates a new thread of execution to perform the operation on behalf of the calling thread. As soon as the new thread has been created, control returns to the calling code, and it may proceed to do other work.

The new thread performs the *DosRead* or *DosWrite* call. When that completes, the new thread signals the event by clearing a semaphore. Then it terminates. When the main thread can do nothing more without the I/O operation, it waits on the semaphore. It will be released from the wait when the I/O is complete. At that time, the code that was returned by *DosRead* or *DosWrite* has been stored and can be examined to see if there was an error.

References: thread 1.2.4; semaphore as signal 9.4.4

8.11.6 Changing the Read-Write Pointer

The read-write pointer of a file is a 32-bit integer that contains the index of the next input or output byte. Only disk files have read-write pointers (or to put it another way, a device's pointer is always zero). The read-write pointer is updated on every read or write by the count of bytes transferred.

The *DosChgFilePtr* call can change the read-write pointer according to a variety of "methods." It takes a 32-bit signed offset value and a method number, and returns the new read-write pointer value that resulted. No physical operation is involved when the pointer is changed, and no significant time is spent doing it. The new pointer setting only takes effect during the next read or write, when the system has to figure out where in the file to go for input or output.

Method 0 simply sets the offset as the new pointer value. Setting the pointer to zero "rewinds" the file to its beginning, allowing it to be read again or overwritten. Setting the pointer to any other value gives the fundamental ability to do random-access I/O.

Method 2 adds the offset to the size of the file. Using method 2 with an offset of zero sets the file to its end, ready to append new output to the existing contents or ready to read the existing contents in reverse order by records. (Why would anyone read a file in reverse order? Well, reel-to-reel tape drives can be read backwards, and some interesting sort/merge algorithms were based on this capability.)

Method 1 adds the offset to the current pointer. Using method 1 with an offset of zero causes the current pointer value to be returned. Using it with a positive offset skips over part of the file. Using it with a negative offset allows some part of the file to be read or written over again. This might be done when a record has to be read and examined before the program knows whether it must be updated.

8.11.7 Forcing Buffered Data to Disk

The system may defer writing disk output in order to get better performance. The buffering, or *cache*, algorithm it uses isn't published and might change from release to release. All we can be certain of is that, unless the Write-through option is specified to *DosOpen*, *DosWrite* only starts data on its way to disk, it doesn't necessarily move it all the way.

As a disk file is extended by sequential output, more disk allocation units are assigned to it. The information about these may also be held in storage for performance reasons.

The *DosClose* operation forces any unwritten output to be written and forces the system to update the disk directory. Making this "commitment" of data to disk at close time is sufficient for most applications. However, there are times when a

program needs to be sure that its output is permanent, secure against system crashes, but it's not convenient to close the file.

The *DosBufReset* call does this job: it commits all file data to disk just as a close would do, but doesn't close the file.

8.11.8 File I/O Performance

File operations are usually the slowest things a program does, and the run time of most programs is proportional to the size of the files they read or write. However, most of the factors that affect the performance of disk I/O are outside the domain of an application program. One, buffer size, is not.

8.11.8.1 Effect of File Location

Seeking, that is, moving the access arm from track to track, is the slowest thing a disk drive does. Next slowest is rotational delay, the wait for the desired sector to rotate under the access mechanism. Neither of these delays can be predicted or controlled by the program. It can't control access time because it can't specify where the contents of a file will be placed on the disk surface. This doesn't matter much for small files, ones less than 50 kilobytes or so, since these will span only a few tracks at most. Nor does it matter much for files that are only processed sequentially, since the access arm will make a smooth progress through the tracks of the file one time and be done. But disk layout does matter for a large file that will be read by direct access.

When a large file is first created, its contents are placed in the first available sectors, and these may not be adjacent. If the disk is newly-formatted, the file will go into consecutive sectors and so will be spread over the least number of tracks. On the other hand, if the disk has seen a lot of use the file may be fragmented over many smaller openings between existing files, and so may occupy a total span of many more tracks than necessary. There is little that can be done to prevent this. One small step would be to make sure that a large file is allocated to its maximum size early, preferably when it is first opened (use the file-size parameter of *DosOpen*). If the file is extended at some later time, the addition will almost surely be allocated some distance away from the initial allocation.

8.11.8.2 The Effect of Arm Contention

In OS/2, disks are serving the I/O requests of many programs at once. It is quite possible that the system will do some disk I/O for one program, then work on a different file for another program, and then return to do work for the first program again. If, between every two disk operations to one file, some other program causes the disk access arm to move to a different track, two seeks will be inserted between every pair of operations. This *arm contention* can have a serious (although unquantifiable) effect on a program's performance.

One way to minimize arm contention is to request disk I/O in large blocks. When a block spans multiple sectors, the system is quite likely to read or write them all in a single burst. That's the most efficient way to use a disk anyway, and it reduces the opportunity for other programs to steal the disk arm.

8.11.8.3 The Effect of Buffer Size

When reading or writing a disk file, the amount of data transferred on each call has a strong effect on the speed of the program. In general, the more data that is transferred and the fewer the *DosRead* and *DosWrite* calls made, the faster the program will go. This is partly because there are fewer system calls and thus less overhead. The largest part of the effect is due to efficiencies in I/O operations.

The data of a disk file is written in discrete chunks called sectors. The system has to transfer whole sectors to or from the disk. The block of data the program wants to read or write (the bytes at the current read-write pointer for the length requested) may or may not align to sector boundaries.

It will never align if the requested length is less than the size of a sector (the size of a sector for a given drive is available via *DosQFSInfo*). When the length is between one and two sectors long, the block may or may not span an entire sector, and its ends may or may not slop over into adjacent sectors. As the requested length goes up, the block spans more and more complete sectors.

When the block being read or written encompasses one or more complete sectors, those sectors are transferred directly between the disk and the program's buffer. This is the most efficient method since the data are not copied, and no extra data is transferred.

The partial-sector scraps must go by a more circuitous route. Since only whole sectors can be read or written, the system has to allocate a sector-sized buffer

elsewhere. It must read the target sector into that buffer. Then, for input, it can copy the desired portion to the program's buffer. For output, it copies the written portion from the program's buffer into the sector, and then it has to write the entire sector back to disk. (Without the Write-through option, it will defer this write in the hope that the next write will touch the same sector.)

Clearly there is more work involved in handling these partial sectors. Extra data (the unrequested parts of the sectors) is transferred through the I/O mechanism. Even when the target sector is already in storage in a system buffer, there is still the work of copying the data bytes.

8.11.9 Locking Files

OS/2 allows two or more programs to share access to a file, even to share write access. If there is any danger of this happening accidentally, it can be prevented or regulated using the Share option of *DosOpen*. However, some applications can be built on multiple access that can be built in no other way. In these applications, a number of programs open the same file with the "deny none" Share option and the intent of both reading and writing in the file.

Chaos would result if all the programs tried to use the same part of the file at the same time. One process might read data that was being changed by another process, and get partly-built or inconsistent data. Or two processes might both decide to update the same record. Each would read it and prepare an update based on what it read; then both would write it back, and the second write would wipe out the changes made by the first write.

The *file lock* mechanism makes it possible to avoid such disasters. It provides a way in which one process can temporarily exclude all other processes from a part of the file. While the others are excluded, the process can read and update the file in peace. Then it can remove the lock and let normal access resume.

The *DosFileLocks* call does both the locking and unlocking. Its parameters are lock *ranges*. A lock range is a pair of 32-bit integers. The first is an offset into the file, the second a length. Together they describe a range of bytes.

DosFileLocks takes a handle and two ranges, one to be unlocked and one to be locked. Ordinarily it would be called only to lock, or only to unlock, and not to do both. A null unlock parameter is passed when only locking is wanted, and a null lock range is passed when only unlocking is wanted. In a regrettable piece of software design, OS/2 requires that a null range be indicated by passing a zero

address on the machine stack. This makes it awkward to declare the *DosFileLocks* procedure in high-level languages. An awkward cast is required in C to generate a zero address. A structured constant of the ADS type has to be used in Pascal, and the parameters have to be declared as ADS types instead of the safer VARS types. (These notations are demonstrated in the include files in the Disk Appendix.)

Once a process has locked a range, no other process may read, write *or lock* a byte in that range. Attempts to do so return the error code for a lock violation. A subroutine to read a record that might be locked would be coded along these lines:

```
while DosRead(hdl,etc) returns 33 do
      DosSleep(0)
```

The *DosSleep* call gives every other process a chance to run. This polling technique is potentially very inefficient. It is only tolerable if the chance of contention is small; that is, if the file has many records and processes access them more or less at random, so that the odds of running into a locked record are small. Even then, a process could burn up a lot of time in this kind of polling if the lock were held for more than a brief time.

If contention may be frequent or locks are held for more than a second or two, the processes should use a system semaphore as a signal that a lock has been released. Then the subroutine to read a possibly-locked record would look like this:

```
while DosRead(hdl, etc) returns 33 do
    DosSemSetWait(central semaphore, maxwait)
```

Whenever a process released any file lock, it would clear the central semaphore. The lock that was released might not be the one that this process is waiting for, but the DosRead will let it know that.

This logic has a flaw: it is possible that the *DosSemSetWait* call could set the semaphore just after the lock holder had cleared it. If that happened, the waiting process could wait forever, or at least until another process had set and released another file lock. The "maxwait" value averts this chance by setting an upper limit on the wait time. It would be a constant approximately equal to the average time a file lock is held in this application.

A process may lock many ranges of a file (just how many isn't specified, but several). However, any application that relies on locking more than one range must be designed with great care to avoid deadlock. Deadlock can arise when processes need to lock two or more ranges each. The simplest case of deadlock arises

when one process succeeds in locking range A but fails to get range B. Meanwhile the process that owns range B is trying and failing to lock range A. If neither gives up the range it holds, neither can ever proceed.

There are many strategies for avoiding deadlock. It's a good rule to always lock multiple ranges in ascending order by offset. Another good rule is that processes must always release all locked ranges before locking any new ones. These rules won't prevent deadlock in all cases, but they make analysis simpler.

DosFileLocks does not require that a lock range be physically part of the file. *DosRead*, *DosWrite*, and *DosChgFilePtr* will not operate on an imaginary file offset, but *DosFileLocks* will, and in fact it will accept offsets that are beyond the size a file can have. That makes it possible to use nonexistent ranges of the file as representatives of other things. For example, the one-byte range at offset 0ffff,fff1 hex might stand for the right to update any part of an index that was scattered through multiple parts of the file. Locking the imaginary range would be easier than locking each part of the index, and might eliminate a possibility of deadlock.

References: DosSleep 9.5.2; semaphore as signal 9.4.4; system semaphore 9.4.5

9

Asynchronous Processing

A *process* is a loaded program and all the resources it owns. A process is a static thing composed of system control blocks and segments in storage; it is the body of a program but not its soul. A *thread* is the spark that animates a process, the soul in its body, the mind in its brain. In this chapter we consider how threads are created and synchronized.

9.1 Threads

In the unpoetic lingo of computer science, a thread is a *dispatchable entity*; that is, it is the thing that the operating system allows to use the CPU for a while. OS/2 manages a large number of threads, and shares the time of the CPU between them. When it gives the CPU to a thread, it *dispatches* the thread; when it takes the CPU back, it *suspends* the thread.

Every process starts with one thread, but may create more. Some OS/2 calls create threads automatically: the *DosReadAsync* and *DosWriteAsync* calls for example. The operating system itself has internal threads.

Reference: asynchronous reads and writes 8.11.5

9.1.1 Dispatching a Thread

In the practical terms a programmer uses, a thread is *a stack and a set of registers*. To dispatch a thread, OS/2 loads up the registers that belong to that thread. One of these is the Local Descriptor Table Register; that defines the address space of the process and all its contents. The registers also include the Code Segment and

Instruction Pointer (*CS* and *IP*); these define the next instruction the thread will perform. The Stack Segment and Stack Pointer (*SS* and *SP*) registers define the call stack, and thereby define the thread's history of called subroutines. The other registers define its working variables, and thus the state of its computation.

In short, when OS/2 loads a thread's register contents into the CPU's registers, the CPU becomes the thread and begins to execute instructions. Since the thread is part of a process, it animates the process and carries out the program that the process describes.

When any interrupt occurs, the registers are saved and control goes back to OS/2. This suspends the thread; since it doesn't control the CPU it can't execute. OS/2 may or may not return to the thread right away. It may choose to dispatch any other thread or threads first.

Reference: registers 2.2

9.1.2 Creating a Thread

Every process starts life with one thread. One thread is sufficient for many programs. Any program originally written for MS-DOS has the assumption of a single thread built into it.

There are many interesting things that can be done with multiple threads within a single process. It takes practice and a new cast of mind to devise multi-threaded programs. Some of the considerations are dealt with later in this topic. Here we consider how threads are created and ended.

The *DosCreateThread* call creates a new thread. It takes two input parameters. The first is the far address of an instruction. Usually the instruction is the first one of a far procedure (a *far extern* function in C, a *public* procedure or function in Pascal, or a *proc far* in assembler). This far address will form the initial contents of the new thread's *CS* and *IP* registers, so the new thread will begin execution at that address.

The second input parameter is the far address of the top (highest address) of the new thread's stack space. This far address will be loaded into the new thread's *SS* and *SP*; that's why it it should be the address of the top of the stack—not the bottom (lowest address) as seems most natural when passing the address of a block of storage. (If you understand the parameter-passing rules of your language very well, it it possible to initialize the new thread's stack with parameters.)

The rest of the new thread's registers will be identical to those of the thread that issues the call. The new thread will have the same address space, of course, but also the same registers. This is an important point. The code of a C or Pascal function makes assumptions about the contents of certain registers, but if C or Pascal code dispatches a subroutine in the same language as a thread, the registers that were right for the running thread will probably be right for the new thread as well.

References: first thread 4.3.1; initial register contents 4.3.2; near and far procedures 2.2.4.2.1; parameter conventions 2.2.3.1

9.1.3 Suspending and Resuming a Thread

Commonly a thread will run as often as it can, and when it must suspend to wait for some external event, it will do so voluntarily with one of the synchronization calls discussed below. However, it is possible for one thread to suspend another. The *DosCreateThread* call returns a number, the *thread ID*. This may be used in a *DosSuspendThread* call to force the thread into the suspended state. A call to *DosResumeThread* will release the thread later.

9.1.4 Ending a Thread

A thread can't be killed by another process—it has to give up the ghost voluntarily with *DosExit*. The *DosExit* call is quite powerful; it can be used to terminate the entire process and pass a completion code back to the parent process. However, it can be used by an individual thread to end only its own existence.

The *DosCreateThread* call pushes the far address of a *DosExit(0,0)* call onto the new stack. As a result, when the code of a thread is a procedure that ends with a far return, the thread may end itself by returning. Thus explicit *DosExit* calls aren't often needed except for special cases.

9.1.5 Thread Priority

OS/2 distributes the use of the CPU among all threads in the system according to a scheme based on *dispatching class* and on *priority*. Dispatching is not based on processes but only on threads (so yes, the process with the most threads will receive the most CPU time in total).

9.1.5.1 Dispatching Classes

There are three dispatching classes, called idle-time, regular and time-critical. A thread in the idle-time class defers to threads in the other classes. A thread in the regular class defers to threads in the time-critical class.

9.1.5.2 Priority Values

Within its class, a thread receives attention based on a numeric priority that varies from zero to 31. The higher the number, the greater the priority and the more likely the thread is to get the use of the CPU. Within a class, threads of equal priority take equal turns to use the machine.

A thread inherits its priority from the thread that creates it. As a result, the threads within a process usually all have the same class and priority. The usual priority, the priority of most threads in most processes, is regular-15.

9.1.5.3 Querying and Setting Priority

A thread may query its own priority with the *DosGetPrty* call. A thread may set its own class and priority with the *DosSetPrty* call. The same call can set the class and priority of all threads within the process to a single value.

Clearly, since threads can set their own priority, the whole system operates on good faith. If you want your program to monopolize the system and lock out all other work, just have it set its class to time-critical and its priority to 31, and let it enter a compute loop! Seriously, the best way to handle thread priorities usually is to not handle them at all, but to leave them alone.

There are two cases where a thread ought to set its own priority. The first is the case of a very lengthy computation. The typical thread gives up control to the system frequently for I/O or to synchronize with other threads. In fact, it's uncommon for a typical thread to use all of its timeslice before suspending for some reason (timeslices are discussed below). A compute-bound thread is different; it always uses all of every time slice, and its presence in a list of other threads of equal priority will cause the others to slow down noticeably. It's a good idea for a thread that will compute solidly for more than a few seconds to lower itself into

the idle-time class. That allows normal threads to operate at nearly normal speed, since idle-time threads defer to regular threads.

The compute-bound thread will not necessarily be delayed by this change! After all, if the computation is important to the operator, he or she will sit back and wait for it to complete, and most other activity in the system will cease. When all time is idle time, an idle-time thread runs at top speed.

The second case is when a thread needs to make a quick response to some external event. Examples are hard to imagine, since quick response is usually needed to respond to an I/O device, and in OS/2 these are usually handled by device drivers driven by hardware interrupts (the quickest response possible). Nevertheless, if a program has to make quick response to some event, it should create a thread and raise that thread to the time-critical class. Then, when the event occurs, the thread will not have to wait its turn to run but will take priority over all regular and idle-time threads.

One thread that can justify the use of time-critical dispatching is the watchdog thread used to redraw the screen for a video graphics application.

9.1.6 Dispatching and Timeslicing

The OS/2 kernel dispatches the thread of the highest class and priority that is ready to run. That thread runs until a thread of higher priority is ready to run, or until its timeslice has expired.

Any of a number of things might occur to make a higher-priority thread ready to run: the running thread might clear a semaphore that a high-priority thread is waiting on, or an I/O interrupt might signal the completion of an I/O operation it was waiting for. Whatever the reason, OS/2 will suspend the current thread and dispatch the one of higher priority.

When no thread of higher priority is ready to run but there are other threads of equal priority, OS/2 tries to share the CPU equitably among them. It does this by giving each thread no more than one timeslice of execution before moving on to the next thread in turn. The length of a timeslice interval is set by the *timeslice=* statement in the configuration file. It is typically the same as the system timer resolution, around 31 milliseconds, so that a timeslice is one timer tick.

9.1.6.1 Minimum and Maximum Slices

Quite commonly there will be only one thread ready to run, and in this case it is wasteful for the system to scan the list of threads for others on every timer tick. The configuration *timeslice* statement allows for setting both a minimum timeslice and a maximum one. The minimum interval is also the normal interval, so long as more than one thread is ready to run. However, when a thread runs for one timeslice and the OS/2 kernel finds that no other thread is ready to run, its next timeslice is lengthened by one timer interval, up to the maximum interval. In this way a thread that is working alone gets longer and longer slices and the system loses less time to the overhead of scheduling.

There is one drawback to stretching the timeslice this way. When another thread of equal priority becomes ready to run after a long delay (long in this context is a few seconds, the kind of delay that might happen while waiting for user input) its start may be delayed by the length of a maximum timeslice. If this is a problem the maximum timeslice can be set equal to the minimum.

9.1.6.2 The Maximum Wait

Once a thread becomes ready to run, it must still wait while threads of higher priority run. In fact, threads in lower classes can be locked out completely if those in higher classes don't sometimes yield the machine to wait for some event.

The *priority=dynamic* and *maxwait=* statements in the configuration file provide a solution. The maximum wait time is stated as a period of seconds. Any thread that waits, ready to run, for longer than this receives a temporary boost in priority so that it will get one timeslice.

This dynamic priority ought not to be necessary. As a general principle, high priority should never be granted to a thread that does a lot of computation. The high priorities should be reserved for threads that must respond rapidly and predictably to some external event (such as a timer interval), but which spend most of their time suspended waiting for that event to occur. When this is the case, threads of lower priority will not be starved, or if they are, it will be because the system is legitimately swamped by high-priority events.

9.1.7 Threads in C and Pascal

It's entirely possible to use *DosCreateThread* in a high-level language. It requires

the ability to allocate a block of storage for a stack, and the ability to form the far addresses of the stack and of a procedure. Standard Pascal doesn't permit forming the address of a procedure, but Microsoft Pascal has been extended with the *ADSPROC* type which meets this need precisely. The code to use the call would resemble that in Table 9.1.

The parallel code for the C language is shown in Table 9.2. (Both of these examples could be simplified by the use of include files from the Disk Appendix of this book.)

While threads can be started from high-level programs, it must be done with care and with some understanding of how the language compiler assumes that storage is laid out. The reason is that the compilers and their run-time libraries have largely been carried forward from MS-DOS. The generated code and the run-time library procedures contain many implicit assumptions that there can only be a single thread in the program.

Take stacks for example. In both of these examples the stack area for the new thread is allocated from the heap, although it might have been a global (static) variable. However, the Pascal compiler always (and the C compiler for all but Huge model programs) places global variables at lower addresses than the normal stack space. Some compiled procedures and most built-in procedures begin with a stack check that compares the stack pointer to the boundary between the stack area and the global variables. If the stack pointer is less, the run-time code assumes that the stack has overflowed and is destroying global variables. That's a reasonable assumption if there is only one thread and thus only one stack!

If the new thread's stack were a global variable, the new thread's stack pointer would start out in the range of global variable addresses, and the new thread would fail the first stack check it encountered. It would be diverted into the code for a run-time error message and would try to end the program. Whether it succeeded or not, things would get ugly.

The examples take advantage of the knowledge that in Pascal and most C memory models, the heap is at a higher address than the stack. The new thread may execute stack checks but they won't detect any problem—not even if it does overflow the allocated size. Although the checks are useless, at least they won't fail needlessly.

Table 9.1
Sketch of DosCreateThread Use in Pascal Program

```
Const
     StkSize = 1024; (* a healthy stack size *)
Type
     Stack = array[1..StkSize] of byte;
     ThreadID = word;
Var
     RC : word;
     ThrID : ThreadID;
     ThrStk: ^Stack;

function DosCreateThread(code:ADSPROC;
                         vars TheID: ThreadID;
                         vars TheStk : byte) : word;
               external;

Procedure [public] Thread;
begin (* code to run as thread here *) end;

begin (* main routine *)
     new(ThrStk); (* put stack in heap *)
     RC := DosCreateThread( ADS Thread, (* code to dispatch *)
                            ThrID,      (* receives thread id *)
                            ThrStk^[StkSize] (* end of stack *)
                            );
     if RC=0 then
           writeln('New thread ID is',ThrID)
     else
           writeln('DosCreateThread returns',RC);
end;
```

Table 9.2
Sketch of DosCreateThread Use In C Program

```
#define STKSIZE 1024 /* healthy stack size */

extern unsigned far pascal DOSCREATETHREAD(
                        void (far *)(void), /* far addr of void fn */
                        unsigned far *, /* thread id */
                        void far * ); /* stack pointer */

static void far thread()
{ /* code to run as thread here */; return; }

main()
{
unsigned rc, thrID;
char *t_stack;

        t_stack = malloc(STKSIZE);
        rc = DosCreateThread( &thread, /* code to dispatch */
                              &thrID,  /* receives thread id */
                              (void *) t_stack+STKSIZE);
        if (rc==0) then
                printf("New thread ID is %d",thrID);
        else
                printf("DosCreateThread returns %d",rc);
}
```

This kind of knowledge is not usually required when writing in a high-level language. But this and more is required in order to run multiple threads, at least until the compilers catch up with the operating system. The problems that crop up are various and subtle; however many arise when two threads make uncoordinated, conflicting changes to variables that they have in common. For example, some compilers generate code for floating-point arithmetic that relies on an accumulator variable in storage. If both threads use floating point, they may trash each other's intermediate answers, with entirely unpredictable results.

Unless your language system promises full support for OS/2 threads, it is probably safest to reserve all floating-point arithmetic to a single thread. It is also best

to adopt a rule of one file—one thread. When two threads operate on the same file, strange things may happen.

These problems arise because the generated code assumes only a single thread. You have to leave that to the compiler writer to fix. However, multiple threads can also collide over the use of program variables. This kind of coordination is your problem; we'll discuss it next.

9.2 Coordinating Asynchronous Threads

Threads usually run as fast as they can. They use the CPU every chance they get and give it up only when OS/2 takes it away or when they must wait for I/O. We cannot predict how often a thread will be allowed to use the CPU, nor for how long each time, nor how long its I/O operations will take to complete.

As a result we cannot predict the progress of any one thread; it depends on everything else that is happening in the system. There is no predictable relationship between the progress of any two threads: their actions are *asynchronous*, that is, unrelated in time.

There is no harm in threads being asynchronous if they share no resources, or if they never modify the resources they share. Threads that don't address the same file or segment of storage may run at any speed; for instance, threads in unrelated processes have no effect on each other. Within a single process, dozens of threads may execute instructions from the same code segment at once because execution can't change the contents of a code segment.

Threads only need to be synchronized when they share a modifiable resource—a file, a device, or variables in storage. Then they must synchronize their work so that modifications happen in a predictable order, and especially so that the partly-finished work of one thread won't be spoiled by the operations of another thread.

Here's an example. Suppose that two threads share code that includes the statement:

```
Counter = Counter + 1;
```

Suppose that at some instant the variable *Counter* contains 27. The first thread executes the machine instructions that load the *Counter* value into a register. Just then a timer interrupt occurs; OS/2 takes over the CPU and decides to dispatch the second thread. By evil chance, that thread happens to be approaching

the same statement. It executes the statement, leaving 28 in *Counter*. Shortly after it passes through again, leaving 29..

Then the timer pops again and OS/2 decides to give the CPU back to the first thread. It completes the statement it was in, incrementing the register and storing 28 into the *Counter* variable. The statement has now been executed three times but *Counter* has been incremented only once; the first thread has destroyed the work of the second one.

Real programs share variables that are more elaborate than integer counters, but this example has the essential features of any asynchronous error. Notice first that the code is correct for a single thread; an error only happens when multiple threads are active. As a result, the bug can't be diagnosed by a compiler, and it will be invisible to the eye of a programmer who isn't thinking in terms of multiple threads.

Notice, too, that the problem only happens when hardware interrupts come at precisely the right times. As a result, the bug will be intermittent, infrequent, and difficult to reproduce. Changes elsewhere in the system that have no apparent connection to the problem will nevertheless affect it because they affect the order and frequency with which the threads are dispatched.

In a way, asynchronous conflict is like a hardware failure. Since no thread can predict the actions of any other, from the standpoint of any one thread, the contents of a shared storage location must seem to vary in a random, unpredictable fashion. If that happened in a single-thread environment, we'd say that there was a hardware bug, probably a bad RAM chip! Errors from asynchronous conflicts have the same kind of difficult, hard-to-catch nature about them.

Asynchronous programs require defensive design. While a thread is working on resources that are unique to it, it is just a program like any other. But in stretches of code that deal with shared resources, you must make the defensive assumption that, between any two machine instructions executed by one thread, *all other threads might execute any number of instructions.* If a bug could arise as a result, you must insert code to synchronize the threads.

Fortunately, threads can be synchronized by any of three methods: critical sections, semaphores as keys, and semaphores as flags.

9.3 Critical Sections

A critical section of a program is a stretch of code that will fail if it is traversed by more than one thread at a time. OS/2 allows you to mark a critical section, and will let only one thread into it at a time. The example above could be made into a critical section like this:

```
DosEnterCritSec;
Counter = Counter + 1;
DosExitCritSec;
```

This facility works in a simple way. When the *DosEnterCritSec* call is executed, OS/2 suspends multithreading within the process. No threads except the one that made the call will be dispatched, and therefore only that thread can occupy the critical code. If it is interrupted, OS/2 may let threads of a different process run, but not threads of this process. Normal multithreading resumes when the *DosExitCritSec* call is made.

A critical section is easy to define and code, and the OS/2 calls have little overhead (they only set a few system flags and return). Within its limits, a critical section is the first choice for synchronizing competing threads. It's an excellent way to safeguard passages of code that update queues, trees, and other pointer-based structures.

The critical section has two failings. First, it is only effective inside one process. Threads of other processes continue to run normally. Therefore a critical section can only protect a resource that is private to a single process; it cannot guard a resource that is accessible to code in more than one process.

Second, a critical section stops all threads but one. If that thread also stops, the entire process will stop. Hence this rule: a critical section must contain nothing that could cause a thread to stop. In particular a critical section should never contain I/O or one of the semaphore waits we're about to discuss. If it did, the entire process would be slowed to the speed of the slowest part of the critical section. And if the wait in the critical section depended on the action of some other thread to release it, a deadlock would ensue—no other thread could run, so the wait would never end.

9.4 Semaphores

A *semaphore* is nothing more than a doubleword in storage—that, and an ironclad convention on how it will be used. So long as the convention is observed, the semaphore can be used to synchronize the actions of two or more threads. A semaphore located in storage that your program can address is a *local* semaphore; it can be used only by code that can address it, which is usually only the code of a single process. (These are called RAM semaphores in the OS/2 manuals.) OS/2 can also allocate semaphores in system-owned storage. Any number of processes can access such a *system* semaphore by giving its name in characters.

9.4.1 Fundamental Semaphore Operations

OS/2 supports three fundamental operations on semaphores and one composite one. The first fundamental operation *sets* a semaphore: it writes the nonzero thread ID of the current thread into the semaphore. After setting, the semaphore is said to be *owned* by the thread whose ID it contains. There is room for only one thread ID in a semaphore.

The second operations *waits*: if the semaphore has not been set it returns at once; if it has been set it suspends the active thread until that semaphore is cleared. Any number of threads may wait on one semaphore.

The third operation *clears* a semaphore: it writes a zero into the semaphore, and it releases all threads that might be waiting, making them eligible to use the CPU once again.

It's important to know that threads released from a wait will get their turns to use the CPU in no particular order. There's no promise of fairness, no rule that guarantees the CPU to the thread that waited the longest. It's pure chance.

These are the primitive operations. The one composite operation is *request*. It tests a semaphore, and if it is clear, sets it. The test and the set are done in system code that can't be interrupted. If the requested semaphore is not clear, the request operation waits on it. After awakening it repeats the test.

These four operations are sufficient to let a semaphore be used in either of two distinct ways: as a key to guard a resource, or as a flag to signal an event.

9.4.2 Semaphores as Keys

A key semaphore is one that is used to guard a shared resource so that only one thread can access it at a time. The semaphore is the key to the resource. By agreement, only the thread that currently holds the key may use it. All other threads must wait at the door until the present occupant comes out and surrenders the key.

In an earlier example several threads needed to increment a variable named *Counter*. Suppose that a four-byte, word-aligned field (a *long* in C, an *integer4* in Pascal) has been declared with the name *CounterSem*, and that a convention is established that *CounterSem* will represent the right to modify the variable *Counter*.

At the beginning of the program *CounterSem* should be initialized to zero, the clear state. Now the code to increment *Counter* may be written this way:

```
DosSemRequest(CounterSem,-1);
Counter = Counter + 1;
DosSemClear(CounterSem);
```

When a thread makes a *DosSemRequest* call, OS/2 checks the semaphore that's passed. If it is presently clear, it is set; the running thread owns the semaphore and therefore (by agreement) has exclusive use of the resource that the semaphore represents. The owning thread continues to run, but all other threads remain eligible to run as well. OS/2 may take control from the owning thread and dispatch another thread in the same or any other process.

If a second thread executes a *DosSemRequest* naming the same semaphore, OS/2 will see that the semaphore is already owned. The second thread will be suspended on that semaphore. We may picture it sitting patiently at a door, waiting for the previous thread to come out and hand it the key. If more threads try to claim the same semaphore they too are suspended.

When an owning thread makes a *DosSemClear* call for its semaphore, OS/2 releases any threads that were waiting for that semaphore, making them eligible to run. All are set to re-execute the *DosSemRequest* call. We can't predict which of them will be the first to run and gain ownership of the semaphore. Perhaps none of them will; a different thread entirely might slip in and grab it. The others will be dispatched in their turns and may or may not be suspended again, depending on what has happened to the semaphore in the meantime.

In this example we are making a key semaphore do the job done by a critical section earlier. One difference between the two is that a key semaphore can be shared among threads in different processes, as we will discuss below.

A second difference is that when one thread claims a semaphore, other threads continue to run. Therefore it is safe to use a key semaphore to guard a passage of code that includes waits. While the owning thread waits, other threads can continue to work.

The fundamental difference is that the critical section guards a passage of code regardless of what it works on, while the key semaphore guards a specific resource regardless of the code that works on it. There are two ways to illustrate this difference. First consider the case that there are several different modifications that may be done to the common resource. If it is protected by critical sections, there must be a critical section around each piece of modifying code. If it is protected by a key semaphore, there is only one semaphore. It could be requested before, and released after each type of modification; alternatively it could be requested once and several modifications done. This gives considerably more design freedom.

A second way to look at this difference is to suppose that there were many identical resources (counters or other variables), and that the code to update a resource was in a subroutine that received the address of the resource to update as a parameter. The subroutine might contain a critical section. However, that would stop all other threads no matter which resource it was operating on.

The alternative is to pair each resource with a its own semaphore, and to pass the pair into the subroutine. The subroutine can claim the semaphore that is peculiar to the resource it will modify. This will delay no other thread unless the other thread needs to modify that particular resource.

When a key semaphore is clear, the overhead of claiming it is very low, a few dozen instructions. When the thread has to wait for the key, there is the overhead of a switch from one thread to another. When there is a lot of competition for a single resource, with many threads waiting often, the number of these context switches can grow rapidly.

9.4.3 Signals by Polling

Concurrent threads often need to inform each other that some event has taken

place. Often the event is nothing more than, "the data you wanted are ready." Even such a simple signal is hard to convey reliably. The problem is making sure that the signal is not missed, while also making sure that it is not seen more than once.

Suppose that we have one thread that produces numbers for a second thread to process. The first thread must be sure that its partner doesn't miss any numbers, but also that it sees each number only once. Our first solution will be based on *polling*. Each thread will *poll*, that is, test or interrogate, a shared variable until it sees an indication that the other thread has done its work. This polling solution is not really useful for any but the simplest applications, but it's easy to follow and sets a pattern we can use later.

The shared variable is an integer in storage called *TheNumber*. The threads agree that when the first thread has a number, it will put it in *TheNumber*, and that when the second thread has finished with a number, it will set *TheNumber* to zero. The first thread proceeds like this:

```
repeat
   TheNumber = the next number;
   while TheNumber <> 0
      DosSleep(0);
as long as necessary.
```

The *DosSleep* call suspends a thread for the specified time in milliseconds. If the time is zero, the calling thread is nevertheless suspended and all other threads of the same or higher priority get a chance to run before control returns. *DosSleep(0)* is thus a way to say "your turn" to other threads.

In the example, the first thread continues to poll *TheNumber* and to give up control until finally its partner has finished processing the number. Here's the partner's logic:

```
repeat
   while TheNumber = 0
      DosSleep(0);
   process TheNumber;
   TheNumber = 0;
as long as necessary.
```

The second thread polls *TheNumber* and uses *DosSleep* to give up control until finally its partner produces a number to process. Then it sets a zero in *TheNumber*, which will permit its partner to stop polling and make a new number.

This scheme will work. It can even be extended to more complicated data than integers by replacing *TheNumber* with *TheAddress*, a pointer to a data structure to be processed. It has two failings. If either thread has a significant amount of work to do while processing a number, the other thread may expend a lot of overhead polling and *DosSleeping*. And, while a thread is polling, it can't be doing any other work.

9.4.4 Semaphores as Signals

A semaphore can be used to signal that an event has occurred. This use is incompatible with the use of a semaphore as a key. A single semaphore cannot be used to represent both a resource and an event.

When a semaphore is to represent an event, the *DosSemSet* call is used to initialize it to mean the event has not occurred. *DosSemClear* is used to signal the event has happened. The *DosSemWait* call is used to suspend a thread until an event occurs.

In our polling example, the threads were really exchanging two signals. The first thread was signalling "a number is ready for you," and the second was signalling "I've finished with that number." Suppose that two semaphores have been set up to represent these events. They might be named *TakeIt* and *Gotcha*. They must have been initialized with *DosSemSet*. The first thread's logic comes out this way:

```
repeat
   TheNumber = the next number;
   DosSemClear(TakeIt);
   (do other things)
   DosSemWait(Gotcha,-1);
   DosSemSet(Gotcha);
as long as necessary.
```

The second thread would work like this:

```
repeat
   DosSemWait(TakeIt,-1);
   DosSemSet(TakeIt);
   process TheNumber;
   DosSemClear(Gotcha);
   (do other things)
```

```
as long as necessary.
```

As far as processing numbers through *TheNumber*, these threads work just like the ones in the polling example. One difference is that, when one of these threads has nothing to do but wait for data, it *does* nothing but wait; it doesn't churn up the system with polling. Another difference is that, as indicated in the pseudocode, either thread could do other work before settling down to wait for its partner.

The *DosSemWait* call tests the semaphore and returns at once if it is clear. If not, the running thread is suspended on that semaphore. When the *DosSemClear* call is made, the thread will be made eligible to run, but it will be positioned to repeat the *DosSemWait* call. If the semaphore doesn't remain clear until the thread runs, it will wait again. That can't happen in the example above. In a more complicated scheme, it might be possible for one thread to clear a semaphore, and for that thread or another to set the same semaphore before a waiting thread had a chance to wake up and see the semaphore cleared.

Sometimes a thread must wait on multiple events. The *DosMuxSemWait* call (*mux* is an engineer's abbreviation of multiplex) permits this; it allows a thread to wait on as many as 16 semaphores and to be told which one was first cleared. This call is not repeated by a thread as it emerges from waiting; it returns control as soon as any semaphore is cleared regardless of whether the semaphore remains clear or not.

9.4.5 System Semaphores

Semaphores can be used to coordinate threads in different processes. There are two ways to do this. First, if the semaphore is in storage that is addressable from both processes, its far address may simply be passed as the parameter of the system calls that use it. This is easy to understand and simple to do when processes share access to the same segment of data storage. (There are several ways to share allocated segments; see the topic on storage allocation.)

Sometimes the only thing that two processes need to share is a semaphore. In that case they do not need to go to the trouble of sharing a segment of data; OS/2 can create *system* semaphores and give access to them by name. System semaphores are used with interval timer services but have other uses as well.

Suppose that different programs might need to update the same disk file simultaneously. One program, perhaps a database manager, creates and manages the file,

but another, perhaps a backup utility or report generator, might be started by the operator and might need to write into the common file. This is just the kind of problem that should be solved with a key semaphore. The semaphore will represent the right to modify the file; all processes will claim it before opening the file, so only one will be able to do so at any time.

Unfortunately the programs don't share any storage and we don't want to complicate their design by making them do so. Let's make it harder: at first we don't know how many programs there will be, or who will write them, or how many files there will be, or what their names are. We will only decree that all the relevant data files must have unique filenames. With that rule the problem can be solved with system semaphores.

System semaphores are created and accessed by character names as if they were files in a directory named *\SEM*. The *DosCreateSem* call makes one if it didn't exist, and *DosOpenSem* locates one if it does exist. Both return a doubleword handle that may be used in any semaphore operation, so once a system semaphore has been located, it may be used exactly like a local semaphore in storage.

OS/2 keeps a count of the number of processes that have accessed a semaphore (*DosCreateSem* sets the count to 1, *DosOpenSem* increments it, *DosCloseSem* or program termination decrements it). The semaphore is kept in existence as long as its reference count is nonzero, so if other programs are using it, it will survive the program that creates it.

With this in mind, the designer of our hypothetical database may decree that before any program writes to a database file, it must create a system semaphore with the same name as the file. If the return code shows that a semaphore of that name already exists, another program is already at work on that file and this program should open the semaphore.

After creating or opening the semaphore, the program can get exclusive use of the file that it stands for with an ordinary *DosSemRequest* call. The return codes of *DosSemRequest* aren't of much interest when the semaphore is in storage. The return code from claiming a system semaphore, however, can show the important fact that the previous holder of the semaphore terminated without clearing it. Depending on the design of the programs, this might mean that the shared resource is in a dubious state and needs cleaning up.

References: filenames 8.1.1; pathnames 8.1.4

9.5 Timer Services

The OS/2 timer services are closely related to the semaphore services and are affected by priorities.

9.5.1 Timer Accuracy

Several factors affect the accuracy of timed intervals and the speed with which a program can respond to them.

9.5.1.1 Timer Resolution

The timer services take intervals that are specified in units of milliseconds. However, the system timer itself is not accurate to one millisecond, or even to several. The fundamental resolution of the timer is available in the global information segment (gotten via *DosGetInfoSeg*); it is typically 31 milliseconds. All intervals must be thought of as being inexact, plus or minus the timer resolution.

Reference: *DosGetInforSeg* 5.1.4

9.5.1.2 Dispatching Delay

When a timer interval expires it generally makes a thread ready to run. However, there is an additional, unpredictable delay between the time when a thread is ready to run and the time it is actually dispatched.

The delay depends on how many other threads in the system are ready to run, and on what kind of work they are doing. If there are many other threads and all are doing compute-intensive work so that they consume their full timeslices, a ready thread might be delayed by several times the minimum timeslice period. It might even be held up for the maximum-wait period, in which case OS/2 will temporarily raise its dispatching priority.

In general, threads of normal priority cannot be sure of responding to a timer (or any other signal) in predictable time. If it is important to respond predictably to the expiration of an interval, the responding thread should be placed in the time-critical priority class.

9.5.2 Delaying With DosSleep

The *DosSleep* call takes a time interval in milliseconds and suspends the calling thread for at least that interval.

When a thread issues the *DosSleep* call, it is voluntarily giving up the use of the CPU in favor of any other thread that is ready to run. Even if the requested interval is zero, every other thread of the same priority will have an opportunity to run before the caller is dispatched again. Threads of higher priority, of course, run whenever they are ready. Threads of lower priority may or may not get a chance to run. They certainly will not run when the *DosSleep* interval is zero, since the calling thread will be seen as ready to run, and will preempt lower-priority threads.

When the requested interval expires, the calling thread becomes ready to run once more, and then will be dispatched in its turn, sooner or later depending on how busy the system is.

9.5.3 Interval Timers

There are two ways in which a thread may mark an interval of time and continue to execute while the interval ticks off. If the interval is singular, not periodic, the *DosTimerAsync* call will start it. A repetitive, periodic interval may be set up with *DosTimerStart*.

Both calls take an interval value and a semaphore. Both signal the expiration of the interval by clearing the semaphore. The thread that started the interval—in fact any thread or threads—may wait on the semaphore.

The difference between the two calls is that *DosTimerAsync* runs just one interval while *DosTimerStart* begins counting another interval as soon as the first one expires, so that the semaphore will be cleared at regular periods (plus or minus the timer resolution).

Either kind of timer may be cancelled by a *DosTimerStop* call. It is a good idea to cancel outstanding timer intervals when the program is terminating.

9.6 A Design Study

Here we will examine a problem of asynchronous design in more detail. The

problem follows a general pattern that will come up frequently in the design of OS/2 applications. Here are the components of the problem.

9.6.1 The Global Resource

First there is a *global resource* of some kind. This might be a file, or a work area in storage, or a hardware device, or some combination of these things. This resource has a complicated structure and lots of internal state information that must be managed carefully. The aim is to make this resource available system-wide to any number of *client processes*.

The basic approach will be to package the resource as a dynamic link library. The library will include segments of code that export functions that clients may call. It will also include segments of data. At least one of these segments will be shared, common to the address spaces of all clients. At least one data segment will not be shared, but will be unique to each client process. The methods of creating dynamically-linked segments of code, shared data, and unshared data are discussed in another topic. Here we'll be using them.

9.6.2 Readers and Writers

Clients will access the resource by calling routines in dynamically-linked code. Some of the access routines will be designed only to inspect the resource, not to change it. When a thread of a client process is animating one of these routines, it is a *reader*. We don't care how many readers access the resource at one time since they don't change it. However, a reader's work may take some time (to search a table or read a file index, for instance), and readers must be assured that the resource won't change while they are using it.

Some of the access routines will be responsible for changing it in some way, adding to or updating its contents. When a thread is executing one of these routines, it is a writer. Readers can't tolerate the concurrent presence of a writer, since the writer would be altering the very data the reader is inspecting. Nor can a writer tolerate other writers; concurrent execution of two writers would create chaos as each changed the other's output. So when a client program is executing in a writer routine, it must have exclusive use of the resource.

9.6.3 Serial Access with a Key Semaphore

The simple way to satisfy these needs is to guard the resource with a single key semaphore. The semaphore would be located in a shared data segment. Any reader or writer must claim the semaphore before using the resource and release it when finished. Out of all the client processes only one thread would be able to use the resource at a time. However, the scheme is simple and reliable, and should be used during the first stages of developing the application.

Later, as more clients want to use the resource concurrently, the key semaphore is likely to become a critical bottleneck in the system. Also it will probably become clear that reader routines are called much more often than writers.

It seems a shame to make the very common readers queue up when they could run concurrently. The mundane parallel that springs to mind is that of a rest room in a public building. The only person who requires exclusive use of the rest room is the janitor who cleans it; the rest of the time it will accommodate as many users as need it. To guard a resource with a single key semaphore only because a writer must have exclusive access, is like allowing only one person into the rest room at a time because the janitor must have exclusive access.

9.6.4 Multiple Readers, Single Writers

What we need is a scheme that lets in as many readers as want entry so long as only readers are present, but which gives writers exclusive access when they arrive. This is possible using the OS/2 semaphore calls. Here's how one scheme works (others can be devised).

The scheme uses two counters and three semaphores, all in a shared data segment. One counter, *RdrCount*, contains the number of readers currently using the resource. The other, *WtrCount*, contains the number of writers who are waiting their turn for exclusive access.

Two semaphores represent signals to readers and writers respectively. *RdrGate* represents the event of "no writers are active." Entering readers wait for this signal to be true (semaphore to be clear). The other, *WtrWait*, represents the event of "no readers are active." Writers wait for this signal.

The third semaphore, *Control*, is a key semaphore that represents the right to modify the resource, including the right to update the counters and set or clear the other two semaphores.

9.6.4.1 Reader Access Logic

The logic of a reader is shown in Table 9.3. A reader begins by waiting on *RdrGate*. Usually that semaphore will be clear and no wait will occur. Then the reader claims the *Control* semaphore. Once it has this semaphore it can be sure that no other thread can alter the resource, including the counters and semaphores. It increments the count of active readers and then releases the semaphore. That permits other readers to follow along and execute the same logic, so there may be many reader threads active at once. (It would be interesting to include code to save the maximum value of *RdrCount*.)

Table 9.3
Gaining Nonexclusive Use of a Global Resource

```
(r0)   DosSemWait(RdrGate, -1)      -- defer to any writers
(r1)   DosSemRequest(Control, -1)   -- get exclusive use
       RdrCount = RdrCount + 1      -- check in as a reader
       DosSemClear(Control)         -- release modify-right

       (body of the reader function)

(r2)   DosSemRequest(Control, -1)

       RdrCount = RdrCount - 1      -- check out as a reader
       if RdrCount == 0 then        -- if last to leave,
           DosSemClear(WtrWait)       -- wake up any writers
       DosSemClear(Control)
```

As we will see, the presence of a nonzero value in *RdrCount* is what keeps a writer waiting. That's what makes it possible for a reader to release the *Control* semaphore and still be sure that the resource won't change while the reader inspects it. When a reader is finished, it must check out by decrementing *RdrCount*. If it is the last reader to leave, it signals that event by clearing the *WtrWait* semaphore.

9.6.4.2 Writer Access Logic

The matching logic of a writer is shown in Table 9.4. It begins by claiming the key semaphore. It immediately sets *RdrGate* to indicate that the event of *no writers active* is no longer true. From that time, any would-be readers arriving at Table 9.3 will be suspended in its first line.

Next the writer notes its presence by incrementing *WtrCount*. There won't be more than one writer working at a time, but there may be more than one writer waiting to work.

Table 9.4
How a Writer Gains Exclusive Use of a Resource

```
(w0)   DosSemRequest(Control, -1)  -- get exclusive use
       DosSemSet(RdrGate)          -- close gate to more readers
       WtrCount = WtrCount + 1     -- check in as a writer
(w1)   while RdrCount > 0 do       -- while any working readers,
           DosSemSet(WtrWait)          -- set sem while holding Control
           DosSemClear(Control)        -- enable Readers to exit
           DosSemWait(WtrWait, -1)     -- wait for last Reader
(w2)       DosSemRequest(Control, -1) -- regain exclusion
       end while

       (body of the writer function)

       WtrCount = WtrCount - 1     -- check out as a writer
       if WtrCount == 0 then       -- if no other writers waiting,
           DosSemClear(RdrGate)      -- waken any waiting readers
       DosSemClear(Control)        -- let readers, other writers in
```

Now, at point *(w1)* in Table 9.4, if there are no readers active, the writer has exclusive use of the resource. Arriving readers will be suspended on the *RdrGate* event at point *(r0)* in Table 9.3, while if one happens to be ready to run at point *(r1)*, it will be suspended when it claims the *Control* semaphore that the writer presently owns. If *RdrCount* is zero, the writer may proceed, knowing that it has exclusive use of the resource.

If *RdrCount* is not zero, the writer must wait for the working readers to finish (the janitor must wait for the last person to leave the restroom). But the readers can't check out while the writer holds the *Control* semaphore; the writer must release it. Once it does so, a number of things might happen. Readers suspended at point *(r1)* may enter the resource and increment *RdrCount*; others suspended at point *(r2)* may decrement *RdrCount* and leave. Other writers suspended at point *(w0)* may enter and increment *WtrCount*.

The writer enters a while-loop. It sets the *WtrWait* semaphore to indicate that the event of *no writers waiting* is false, releases the *Control* semaphore, and waits for the event of *no readers active* to become true. We've seen how that is signalled by the reader that finds it has reduced *RdrCount* to zero. When the writer wakes up from this wait, at point *(w2)*, it reclaims the *Control* semaphore and returns to the top of the loop at point *(w1)*. The statements made two paragraphs back are true once more.

Why is this a loop? At first glance it would seem that one wait for the *RdrCount* to go to zero would be enough. Not so! Although arriving readers will block on *RdrGate*, there could be *any number* of readers that just happened to be ready to run at point *(r1)*. It could happen that one reader will finish, reduce *RdrCount* to zero, and clear *WtrWait*—and that then another reader, one that OS/2 had just happened to suspend at point *(r1)*, will begin running, increment *RdrCount*, and be suspended again. Thus it is possible that the awakening writer may find that the *RdrCount* isn't zero after all. This isn't *likely*, but in code that will be executed billions of times, a million-to-one chance is certain to occur several times (usually during a demonstration to an important customer).

After the writer finishes its work, it checks out by decrementing *WtrCount*. If it proves to have been the only active writer, it clears *RdrGate*, signalling the event *no writers active* and releasing any waiting readers (the janitor takes down the out of service placard from the restroom door).

9.6.5 Failures and Catastrophes

This scheme is vulnerable to failures in the reader and writer code. A reader or writer might be terminated in the midst of its work by any of several catastrophes. There might be a bug in the code, of course, but there are also factors outside the code. The operator can hit Control-Break, for instance. That's like a lightning stroke; it ends the program wherever it happened to be. The odds of its striking within the resource code can be minimized by making that code run

quickly, but sometimes that isn't possible. If resource segments are swapped to disk by the system, the shortest code sequence may take appreciable time.

If a client program does file I/O, whether as part of resource access or in an unrelated, asynchronous thread, there is always a chance of a hardware problem occurring. Then the operator will see a pop-up menu on which one choice is "terminate the program." That's another way a client could die in the midst of executing a reader or writer function.

If a reader terminates without checking itself out, *RdrCount* will be left nonzero, and writers will wait forever. If a writer terminates without checking out, it will leave *Control* and *RdrGate* uncleared and *WtrCount* nonzero, so readers and other writers alike will wait forever. Worse, of course, is the chance that the dying writer left the resource in a half-finished state.

These are all rare, chance occurrences, but it's worth repeating that, if code is executed often enough by enough clients on enough machines, sooner or later everything that *can* happen in it, *will* happen.

9.6.5.1 Using the Exit List

All these problems could be answered if only a dying program could be given a chance to arrange its affairs properly before it passes on. OS/2 provides for this with the Exit List. The mechanism of setting up an Exit list is discussed in another topic. Here we'll sketch how it can be used in the context of our design.

Through the use of the *DosExitList* call, the resource management code can ensure that an exit routine of its choice will be called when the client program is about to terminate. When the exit routine is entered it must have some way of knowing whether its process was an active reader or writer at all. The odds are that it was neither, and in that case the exit routine has nothing to do and should return to OS/2 at once. We'll consider how the exit routine can know this later.

Reference: *DosExitList* 5.3.3

9.6.5.2 Cleaning Up a Writer

When the exit routine discovers that its dying client was an active writer, its major concern is that the writer may have only partly finished an update to the common resource. If so, that partial work must be removed or fixed up in some

way so as to leave the resource accurate and usable by other clients. How this will be done depends on the details of the application. It is likely to be a challenging design problem—one that will be *much* easier when every update operation is designed from the start with the understanding that it might be broken off and require cleaning up.

Once that difficult problem has been handled, the exit routine for a writer need only clean up the semaphores and counters by executing the last four lines of Table 9.4.

9.6.5.3 Cleaning Up a Reader

If the dying process was in the code of a reader, the exit routine only needs to decrement *RdrCount*. It appears that all it needs to do is execute the last five lines of Table 9.3. That isn't the whole story, however. An exit list routine is called by OS/2 when a *process* is being terminated. But a process might have had more than one *thread*. Since our design permits multiple readers, there is the chance that this dying process might have represented *more than one* reader; if it did, the exit routine needs to decrement *RdrCount* by more than one. We will have to keep track, not of whether a client process is a reader, but of *how many* readers it is. We'll return to the method shortly.

Exit routines are supposed to be quick. In order to reduce *RdrCount*, the exit routine must claim the Control semaphore. There may be a delay here, and we must ensure it isn't indefinite in length. The exit routine knows that there was an active reader (the process it was called for); therefore there cannot have been an active writer. Therefore the only contention for the Control semaphore will be from other readers and from writers entering the top of Table 9.4, neither of which holds the semaphore very long. Just the same, this use of *DosSemRequest* should pass a finite timeout value, 500ms perhaps, and the return code should be tested. In case of a timeout, the exit routine has to give up.

9.6.5.4 Per-Client Information

These plans all presume that an exit routine can tell the state of its process with regard to the common resource: was it animating a writer function, reader functions, or was it doing something unrelated? An exit routine is called when a process is terminating, so this information is kept on a per-process basis. It isn't global to the entire system (there may be other client processes that are running, or terminating, independently), nor is it particular to a specific call into the

common code, since that's done by a thread and there may be more than one thread in a process.

It isn't difficult to keep information unique to a process. It's an option of any dynamically-linked data segment whether it should be common to all processes that use it, or should be recreated in a unique copy for each client. The semaphores and counters must be common to all, but three items need to be kept in a segment that is unique. One is a flag that's true while (a thread of) the process is the working writer; another is a count of how many threads of the process are working readers. The last is a flag that shows whether or not an exit list entry has been made for the process as yet.

The resource package may require all clients to call a general initialize function. In that case, setting up an exit routine would be a part of a larger initialization, and the last flag would reflect whether this client has been initialized yet or not.

The logic that maintains these items for a writer is sketched in Table 9.5. This code replaces the central line of Table 9.4, expanding on body of the writer function. At that point, there can be only one writer in the system (let alone in the process).

The comparable logic for a reader is outlined in Table 9.6. This replaces and expands on the body of the reader function in Table 9.3. At this point the reader has enrolled as a reader, but it cannot assume it's alone. Other threads may also be animating a reader function, and in fact, other threads of the same process may be doing so. Threads of different processes don't contend for the same per-client data segment, since every client process has its own copy. But threads of the same process could conceivably try to update the same segment concurrently. That's just the situation that the critical section was meant to handle. It freezes out all other threads of the same process, so that this thread can update the per-process segment in peace. Meanwhile, reader threads from other processes may continue to run.

Finally, the logic of the Exit routine is sketched in Table 9.7. It does not need to use a critical section since, by the time OS/2 calls an exit routine, a process has only one thread left and it is used to call the Exit routine.

Table 9.5
How a Writer Sets Per-Client Information

```
if Per-client-Init == false
     DosExitList(1,addr(ExitRoutine))
    Per-client-Init = true
Per-client-Am-Writer = true

(body of writer function)

Per-client-Am-Writer = false
```

Table 9.6
How a Reader Sets Per-Client Information

```
DosEnterCritSec
if Per-client-Init == false
     DosExitList(1,addr(ExitRoutine))
    Per-client-Init = true
 increment Per-client-readers
DosExitCritSec

(body of reader function)

DosEnterCritSec
 decrement Per-client-readers
DosExitCritSec
```

Table 9.7
Logic of Process Exit Routine

```
if Per-client-Am-Writer then
     (writer-exit logic as usual)

elseif Per-client-readers > 0 then
        if 121 <> DosSemRequest(Control, 500) then
                RdrCount = RdrCount - Per-client-readers
                if RdrCount == 0 then
                    DosSemClear(WtrWait)
                DosSemClear(Control)
        endif
endif
DosExitList(3,null-address)
```

9.6.5.5 Incompleteness

There is a hole in the logic shown in the last two tables. In fact there are several of them. There are two intervals in Table 9.3 during which a reader owns the Control semaphore; what if it is terminated there? There's no provision for that case in the Exit routine.

There are two brief intervals in Table 9.6 (above the top line and below the bottom one) when it has been counted in the global *RdrCount* but not in its per-client reader count. At the same boundary points in Table 9.5 a writer owns the Control semaphore and is counted in *WtrCount* but hasn't updated its per-client writer flag. In any of these cases, unexpected termination will not be cleaned up.

You may try to improve this scheme; there is room for improvement. But there is a fundamental problem: there is always an interval of some size between the statement that creates a situation and the statement that documents that situation for the Exit routine. If termination occurs in that interval, there's nothing to be done about it. The only solution would be some kind of global critical section so that the two statements could be made indivisible. If it is really important, a critical section of this kind can be created using code with I/O privilege and setting and resetting the interrupt-enable flag. For most applications these intervals

can be made so narrow that the chances of termination in them are really millions to one, and the biennial, inexplicable failure that results is acceptable.

9.6.5.6 Summary of Design Study

This study has remained at a very high level; it ignored most of the details that make such a package interesting and difficult to build. However, the pattern laid out here will serve a variety of applications, and it revealed a use for almost every synchronizing tool OS/2 offers: it has employs semaphores as signals, semaphores as keys, and critical sections all in one package.

In the disk appendix to this book there's a simple package that follows the pattern discussed here. The common resource it manages is a system-wide pool of named variables, a global environment. It uses all the logic sketched in the tables in this topic, including an Exit routine.

9.7 Queues

A *queue* is a data structure that is a staple of beginning courses in computer science. It's a sequence of items that are ordered according to some rule. Since the items are ordered, one must be the head of the queue. A queue must support at least the two operations of adding a new item and of removing the current head item.

The OS/2 kernel contains services to create queues, to put items into them, and to retrieve items from them. These queues are designed to be used for communication between different threads (many threads may add and remove items concurrently without special serialization) and between threads in different processes.

In use, one process creates a queue, giving it a character-string name like the name of a system semaphore. Other processes may gain access to the queue by opening it. Threads in any of the processes may put items into the queue. Threads in the process that created the queue may take items off the queue.

9.7.1 Creating a Queue

A queue is created by the *DosCreateQueue* call. This call takes two parameters: a queue name and a choice of discipline (the rule under which the queue is ordered.) It returns a handle that can be used for writing or reading the queue.

The queue's name is a null-terminated character string that has the form of a filename in an imaginary directory *\QUEUES* (compare to system semaphores, which are named as if they were in the directory *\SEM*). This name is not private to the process; it will be known system-wide so that other processes can open the queue.

The queue must operate under one of three disciplines. The simplest is First-In-First-Out (FIFO). The items in a FIFO queue are ordered by their time of arrival in the queue, just like the people who wait in a queue for service, so the head item is the one that arrived first and has waited longest. The queue may operate under a rule of Last-In-First-Out (LIFO), under which the head item is the one that arrived most recently (a stack is a LIFO queue).

Or a queue may be ordered by priority. In this case, a priority number must be specified whenever an item is added to the queue. Items with higher priority numbers enter the queue nearer its head, while items of equal priority are kept in FIFO order.

The process that creates a queue may add items or remove them, in any sequence, and from any number of threads. Thus the queue can be a means of communicating between asynchronous threads of a single process.

References: filenames 8.1.1; pathnames 8.1.4

9.7.2 Opening a Queue

A queue's name is know throughout the system. Any program that knows the name may open the queue. *DosOpenQueue* is used for this; it returns a handle that may be used only for adding items to the queue. Only the process that created the queue may remove items.

OS/2 presumes that one process creates a queue in order to offer a service. Using *DosCreateQueue* is like hanging up a sign, "Open for business." Other processes request services by putting items on the queue. If other processes were free to remove queue items, they could interfere with the queue, or could snoop into the flow of service requests. So long as only the creator can remove items, a single queue can be a secure channel to many customers.

When a queue is found and opened, *DosOpenQueue* returns the process ID of the process that owns the queue. This number might be used to send a signal to the owning process with the *DosFlagProcess* call.

Reference: sending a signal 5.3.1.1

9.7.3 The Form of an Item

The *DosWriteQueue* call adds a new item to a queue. A queue item consists of four words of data that are usually referred to as a description, a size, and a far pointer. The words might have these uses. When they do, each queue item is a block of storage of variable size. The description word specifies the format of the item, the size word gives its size, and the far pointer gives its address.

In fact, the four words of a queue item may have any uses that the communicating programs agree on. OS/2 doesn't examine or censor the contents of an item in any way. If two processes want to pass 64-bit floating point numbers through queue items (and are willing to split them into three procedure parameters), they may do so.

9.7.4 Passing Addresses

The conventional form of a queue item assumes that a far pointer can be passed from sender to receiver. That will be true when sender and receiver are threads in the same process, but will often be false when the sender and receiver are in different processes. Each process has its own address space, so a selector number from one process will be irrelevant in the context of another process (will select no segment at all, causing a hardware trap and termination, or worse, will select an entirely different segment of data). Some segments are shared between processes but can have different selector numbers in different address spaces. Only a data segment linked as part of a dynamic link package, or a data segment shared with the *DosGetSeg* call, is sure to have the same selector in every address space.

It is possible to pass a mere offset, so long as the receiver can figure out the proper selector to use with it. For example, if both programs are sharing two named segments (allocated with *DosAllocShrSeg*), an offset can be passed along with a word of 1 or 2 to indicate which shared segment is meant.

References: *DosGetSeg* 7.4.2; *DosAllocShrSeg* 7.4.1; selectors for shared segments 2.4.6

9.7.5 Scanning a Queue

The queue's creator may examine the items in the queue without removing them. This is done with the *DosPeekQueue* call. This call takes a number and returns a copy of one of the queue items. If the number is zero, *DosPeekQueue* returns a copy of the head item.

It also returns a number that is some kind of index to the item it returned. If that same number is passed on the next *DosPeekQueue* call, the following item (in queue order) will be returned, along with its number. The result is that *DosPeekQueue* can be used to examine the entire queue.

At least, that's the idea, and it will work well enough for a queue ordered by the FIFO rule. However, items are added to a LIFO queue at the head, and they are inserted into a priority queue at any of fifteen points. If an item is added to the queue at a point closer to the head than the last item peeked, the new item won't be seen until the scan is started over. The whole idea of scanning a queue should be discarded when the discipline is not simple FIFO. (Besides, the only purpose in scanning the entire queue would be to remove an item out of its queue order. If that's necessary, why have an ordered queue in the first place?)

The position numbers returned by *DosPeekQueue* are not simple ordinals. They are probably offsets within a segment that was created to contain the queue, but we aren't told. Since we don't know, we should not assume anything about them. We should not assume, for instance, that there is any arithmetical relationship between the position numbers of two items.

9.7.6 Reading a Queue Item

A queue item is removed from the queue with the *DosReadQueue* call. This call takes a number; if it is zero, the head item is removed and returned. The number may also be a position number returned by *DosPeekQueue*. Since *DosPeekQueue* copies the entire queue item, the only reason to Read after Peeking is to remove the item from the queue.

Both Peek and Read return the Process ID of the process that sent the item. This number may be used to signal the sender with a *DosFlagProcess* call.

9.7.7 Querying and Purging Queues

Any process that has opened a queue may inquire how many items it contains; this is done with *DosQueryQueue*. A process could use this call to find out how much work the server had to do before it would get around to the next request. The server might take a high number of queued items as an indication that it ought to raise its own dispatching priority.

The server can cause the contents of the queue to be completely purged. The *DosPurgeQueue* call is not essential; its effect can be produced by applying *DosReadQueue* in a loop until it returns a nonzero error code. Purging the queue is a drastic action that would be done only by a program that was taking radical action to recover from an error.

10

Video Output

This chapter is about putting output on the display screen for the operator to see. The discussion is written for programs that will be called from the OS/2 command shell and *not* from within the OS/2 Presentation Manager.

The first topic is a review of how the display hardware works and how it is shared among multiple programs; then a note on how the display can be driven very successfully using a stream of data containing ANSI escape sequences. After that we dig into text-mode operations and finally into graphics.

10.1 Display Fundamentals

The section of storage between physical addresses a0000h and bffffh is dedicated to the use of the display hardware. The hardware continually scans this *display buffer* from end to end in synchrony with the electron gun in the video monitor as it sweeps out its raster pattern on the screen.

10.1.1 The Display Buffer

What the hardware picks up from the display buffer determines what will be painted on the screen. The fundamental method of display output is to store data into the display buffer. Under MS-DOS, that is usually done by the ROM BIOS, but some MS-DOS programs update the display buffer directly. OS/2 programs run in protect mode and do not usually have any way of forming a physical address such as that of the display buffer. Furthermore, OS/2 lets multiple programs run concurrently and shares the display hardware among them. Remark-

ably, it is still possible for properly-written programs to operate directly in the display buffer.

10.1.2 Display Modes

At any time the display hardware is operating in one of a few *display modes*. The current display mode determines which parts of the display buffer will be used. It also determines how the hardware interprets the data in the buffer. The contents of the buffer have radically different formats in different modes.

10.1.2.1 Graphics Modes

There are two categories of display modes: text and graphics. In a *graphics mode* the hardware treats the display screen as an array of points called *pels* (from *picture elements*, sometimes called *pixels*). The color of a pel is specified by a single bit or a group of two or four or eight adjacent bits in the display buffer. In enhanced graphics modes, the display hardware folds groups of four bits into single bit positions, making a very complicated and difficult addressing scheme.

OS/2 provides very little support for graphics display modes. It offers a scheme by which a program may manipulate the display buffer and hardware without conflicting with other programs, and that's about it. The operation of the graphics display is up to the program.

10.1.2.2 Text Modes

In a *text mode* the hardware treats the display screen as an array of rectangular character boxes. The contents of a character box is specified by a 16-bit *character cell* in the display buffer. Each cell contains an 8-bit character and an 8-bit attribute byte.

The character boxes have a height and width measured in pels. A box has a *background* attribute and a *foreground* attribute. The background attribute controls the color of the box, while the foreground attribute controls the color of character that is displayed in the box. The attributes also control intensity and whether the character will blink. Each attribute is specified by four bits of the attribute byte.

OS/2 offers considerable support for text display modes. ANSI escape sequences provide device-independent control of the cursor and choice of colors. If more control is needed, there are calls for reading and writing character cells singly, in replicated groups, or in strings at any point on the screen, and for scrolling rectangular patches of any size in any of four directions. If those aren't enough, a program may update the logical display buffer for its screen group directly.

References: screen attributes 10.4.2; ANSI sequences 10.3.4

10.1.3 The Hardware Generations

The display modes are defined by the hardware. Each generation of display hardware has supported the modes of the generation before, and added modes of its own.

The modes of the original IBM PC are still supported by all systems; they form a least common denominator for displays. The IBM PC had two different display adapters (attachment cards): the Color Graphics Adapter (CGA) and the Monochrome Display Adapter (MDA).

The IBM Enhanced Graphics Adapter (EGA) was first supplied with the IBM PC/AT. It supported the modes of the CGA and MDA, and added several new modes of pixel graphics as well as the ability to define new text fonts.

The IBM Personal System/2 computers contain a Video Graphics Array (VGA) that supports all the modes of the EGA and adds yet more graphics modes.

An OS/2 program can still find itself running in hardware that supports only the display modes of the CGA or the MDA. Programs that need another function should test to make sure that the function is available. The method of doing this is covered below.

10.2 Screen Groups

It is important to distinguish between the physical display hardware and the imaginary, unreal display that is part of an OS/2 *screen group*. A screen group represents a virtual operator interface, including a keyboard, a mouse, and a display. There is only one physical display in the system, but there are sixteen screen groups.

At any moment just one screen group is connected to the physical hardware. The operator can see the display output produced by the program (or programs!) running in that screen group. Displays produced by programs in other screen groups are not visible, even though the programs may be running and producing new output.

The display hardware in some modes supports multiple *pages*, independent display areas within the display buffer. The virtual display in a screen group does not embody any concept of pages; it has only one page of displayed text.

10.2.1 Switching Screen Groups

The operator chooses which screen group to watch, and when. The programs cannot dictate which screen will be visible (except that the *pop-up* screen has priority); nor can they predict when the operator will choose to look at their screens.

When the operator presses the hot-key combination to bring up the next screen group, OS/2 has to save the state of the display hardware and then set the hardware to display the output of the incoming group. In some cases OS/2 can do this without the help of the programs that created the displays. Programs that build graphics screens, however, have to assist the system in switching the display. The point to keep in mind throughout the section is that the display hardware is shared among many programs. The more detailed the control a program needs over the hardware, the more it must take this sharing into account.

10.2.2 Is a Screen Available?

Before a program tries to use the display, it should test to see if it has a display available to it. A program started with a Run command in the configuration file is not part of any screen group and has no normal access to a display screen. The same is true of a program started by the Detach command, and of any program that is a child process to either of these kinds of programs.

A detached program can still display screen output through a *pop-up*. Pop-up displays allow a program to preempt the display to show the operator a text-oriented display of an error message or some other urgent information. They aren't practical for more than a brief interaction.

How can a program tell if it has access to a display? Probably the simplest test is to use the *VioGetANSI* call to sample the display. If this call returns any

nonzero code the program probably has no screen group. (Code 465 explicitly means "a detached process has no access to a display.") Another test is discussed in the chapter on Keyboard input.

If the program depends on the display but has no access to one, it should issue a rueful message to the standard error file and terminate.

References: pop-up displays 10.4.4; testing for a keyboard 11.1

10.3 Stream Output to the Display

The simplest way to control the display is to treat it as a file and write a stream of ASCII text to it. There are many advantages to this method. It is hardware-independent, and so relieves the program of any need to discover the display configuration or its current mode. It uses only the *DosOpen* and *DosWrite* system calls, so the program needn't contain the many procedure and structure declarations of the "Vio" series of calls described below. And it is fully compatible, both downward with MS-DOS and upward with the Presentation Manager.

References: *DosOpen* 8.7; *DosWrite* 8.11.3

10.3.1 Standard Output to Display

Every program starts out with a standard output file at handle number 1. This file is often directed to the display, and a stream of output directed to standard output will usually be displayed. This isn't a certainty, of course. Standard output may have been redirected to a file or to the null device.

How can the program tell if standard output is the display? Use the *DosQHandType* call to get the device driver attribute word related to handle 1. If bit 1 is on in this word, signifying the standard output device, the file is connected to the display. If bit 0 is off, standard output is a file or possibly the null device.

A program that uses the ANSI escape sequences described below could use this test to decide whether or not to include escape sequences in its output.

References: standard handles 8.4; *DosQHandType* 8.10.2

10.3.2 Opening a File to the Display

Regardless of the destination of standard output, a program may create a file-oriented interface to the display by using *DosOpen* to open a file to the name of the display device driver, *SCREEN$*, and using *DosWrite* to send strings of data.

It may be possible to open a file to the display using the file operations of a high-level language, for instance with:

```
disp_hdl = fopen("SCREEN$","w");
```

in C, or with:

```
assign(ATextFile, 'SCREEN$');
rewrite(ATextFile);
```

in Microsoft Pascal. This is a fine way to create a stream of display output except for one thing: output to such a file must go through the text output procedures of language involved. These procedures eventually reach *DosWrite* but they may impose a severe overhead penalty as compared to direct calls to *DosWrite*. (Compare, for example, the output speed of the More command, written in C, to that of the Type command which uses *DosWrite* directly.) Also, the language routine may interpret some output byte values by its own rules.

References: *DosOpen* 8.7; *DosWrite* 8.11.3

10.3.3 Teletype-style Output

Stream output is usually sent to the display via *DosWrite*, passing a file handle that is open to the screen. This is the best way to write display output when there is any doubt whether the handle is open to the screen. *DosWrite* will operate correctly with the screen, a file, or a pipe, whereas the *VioWrtTTY* call works only on the screen.

When the screen is the only possible destination, stream output may be sent directly to it using the *VioWrtTTY* call. This is the call that *DosWrite* uses for screen output. Its job is to display ASCII text on the screen beginning at the current cursor position.

VioWrtTTY does three things differently from other display output calls (discussed below). First, it begins output at the current cursor position and up-

dates the cursor position as it goes. Other calls take a row and column starting position as a parameter and don't move the cursor.

Second, it interprets the output data as a stream of ASCII codes. In particular it looks for the codes signifying return, linefeed, backspace, and horizontal tab, and takes these as commands to move the cursor and scroll the screen. Other display output calls put those byte codes in the display buffer, producing graphic symbols on the screen. *VioWrtTTY* also watches for and interprets the ANSI sequences discussed in the next topic.

Finally, only *VioWrtTTY* respects the printer-echo toggle, echoing output to the printer if Control-PrintSc (or Control-P) has been pressed. Other display output calls ignore it, and their output is not echoed to the printer regardless of the toggle's state.

10.3.4 ANSI Screen Controls

A stream of display output sent through the file system (or through *VioWrtTTY*) will be ASCII text, but that needn't prevent the program from choosing colors and controlling the cursor. The support for ANSI (American National Standards Institute, the body that established a standard for video terminal escape sequences) escape sequences loaded as a device driver in MS-DOS, is built into the OS/2 display output call named *VioWrtTTY*. Provided that ANSI support hasn't been switched off by a call to *VioSetANSI*, the program can command many display functions through ANSI escape sequences.

10.3.4.1 ANSI Syntax

The ANSI escape sequences detailed in Table 10.1 can be embedded anywhere in the stream of text.

The form of the ANSI sequences may seem strange at first, but is quite logical when understood. Each sequence begins with the ESC code (byte value 27, or 1bh) followed by a left-square-bracket [(byte value 91, or 5bh). Whenever this two-byte prefix appears it will be swallowed and not displayed, whether the sequence turns out to be a valid ANSI one or not.

Table 10.1
Escape Sequences Supported by ANSI.SYS

Sequence	Purpose
`H`	Move cursor to row 1, column 1 (upper left corner).
x`;`*y*`H`	Set cursor to row *x* and column *y*.
x`A`	Move cursor up *x* lines.
x`B`	Move cursor down *x* lines.
x`C`	Move cursor right *x* cells.
x`D`	Move cursor left *x* cells.
`2J`	Clear screen, home cursor.
`K`	Clear from cursor to end of line.
`s`	Save cursor position.
`u`	Restore saved cursor position.
`0m`	Reset attributes to white on black.
`1m`	Set high intensity on.
`4m`	Set underscore on (MDA only).
`5m`	Set blinking on.
`3`*c*`m`	Set foreground text color to *c*.
`4`*c*`m`	Set background text color to *c*.

Color digits *c* are:

0	black	4	blue
1	red	5	magenta
2	green	6	cyan
3	yellow	7	white

Sequences may be embedded at any point in text written with VioWrtTTY. Each sequence begins with the two bytes ESC (01bh) "[" (5bh).

The ANSI prefix may be followed by numbers that will be parameters to the sequence. Each number is given as digit characters (not binary numbers or byte values). Successive numbers are separated by semicolons. Some sequences expect no numbers, others one, others two, and one will take many numbers. Since the meaning of the sequence isn't known at this point, the *VioWrtTTY* code will set aside any sequence of digits and semicolons and not display it—even if the sequence eventually turns out to be invalid.

The sequence ends with a single nondigit character that determines its meaning and the use of the preceding numbers. We might call this letter the verb; like the German language, "the ANSI standard the verb at the end puts!"

Some verbs take no numbers (*s* and *u* for example). Some, like *A* through *D*, use only the first number. Some require fixed values; *J* for example only works as *2J*. (The full ANSI standard specifies a number of functions for verb *J* but OS/2 only implements the function for a parameter of 2.) The *m* sequence that sets display attributes will accept as many numbers as were given and do each in sequence. Thus the sequence *5;42;31m* sets up for red symbols on a blinking green background.

10.3.4.2 Use of ANSI Sequences

Table 10.1 is not the whole story. There are also ANSI sequences for setting the display mode. However, only some of the possible modes are supported, and there is no sequence to report the current mode. The *VioGetMode* and *VioSetMode* calls described below are preferrable.

There is an ANSI sequence (6n) that requests a report on the cursor position. An ANSI sequence (*x*; *y*R) is inserted into the standard input stream in response. (It isn't clear whether this works when standard input is open to a file or pipe.) The *VioGetCurPos* function is much more convenient.

There is no ANSI sequence for cursor shape; the *VioGetCurType* and *VioSetCurType* calls can be used for that.

The *VioGetANSI* call returns a 1 if ANSI Support is active. If it returns a zero, escape sequences won't be effective. They won't cause any special harm, but they'll appear on the screen mixed with the information that was meant for display, and the cursor won't be positioned as the program expected.

ANSI support can be disabled and later reenabled with calls to *VioSetANSI*. ANSI support might be disabled for debugging purposes, so that escape sequences were visible on the screen.

There is some, but not much, processing overhead involved in using ANSI sequences: a few hundred instructions per sequence at most (not to be confused with the overhead imposed by a language's file-output procedure). If the ANSI sequences give enough control, this small overhead cost is an excellent tradeoff for the extra complexity of managing the screen buffer directly.

10.4 Direct Display Output

Stream output may not allow enough control over the screen. It passes through *VioWrtTTY*, and that call interprets the ASCII format characters as controls. Thus output of a byte *0dh* through *DosWrite*, for example, will not display a musical note symbol; it will return the cursor to the left margin. Nor does it afford a way to scroll smaller areas than the whole screen, or to change cursor shape. For these reasons you might need to gain more intimate control over the display. That's especially likely if you are converting an application that runs in MS-DOS and relies on the easy access to screen hardware that MS-DOS allows. Direct display output provides that tighter control, or as much of it as is feasible in a multiprogramming system. There are three parts to the OS/2 support for direct display output. There is a set of calls for learning the state of the display hardware and perhaps changing the state. Another set of calls provides for text-mode output, and a third set supports graphics modes.

10.4.1 The Display State

The display hardware is controlled by a number of software and hardware parameters that constitute its state. There are OS/2 calls that return these parameters, and matching calls to alter them. The full list of calls and the subjects they deal with, are shown in Table 10.2.

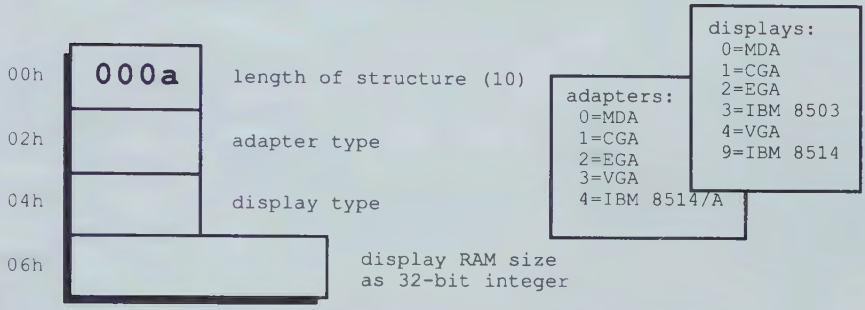
Table 10.2
Summary of Vio Functions for Display State

Inquiry Function	Change Function	Subjects dealt with
VioGetConfig	*n.a.*	Monitor type, adapter type and RAM
VioGetANSI	*VioSetANSI*	ANSI escape sequences
VioGetCurType	*VioSetCurType*	Cursor shape, visibility
VioGetCurPos	*VioSetCurPos*	Cursor position
VioGetMode	*VioSetMode*	Graphics or text, screen dimensions
VioGetState	*VioSetState*	Palette registers, border color, attribute bit 7
VioGetFont	*VioSetFont*	Character symbol bit maps
VioGetCP	*VioSetCP*	Codepage number

10.4.1.1 The Display Configuration

The *VioGetConfig* call returns what OS/2 thinks is the display hardware configuration in the data structure shown in Figure 10.1. The system determines this information while booting up.

Figure 10.1
Structure used in VioGetConfig and VioSetConfig



This is the basic information needed by a program that requires a certain level of display hardware. However, the numbers shown are not a closed set; higher numbers will surely be defined. Therefore the program should always be prepared to accept a number that is equal to *or greater than* the number of the adapter it needs. If the program requires a graphics mode supported by the CGA it should accept any adapter number greater than zero; if it requires EGA features it should accept any number greater than one since the VGA supports all EGA modes.

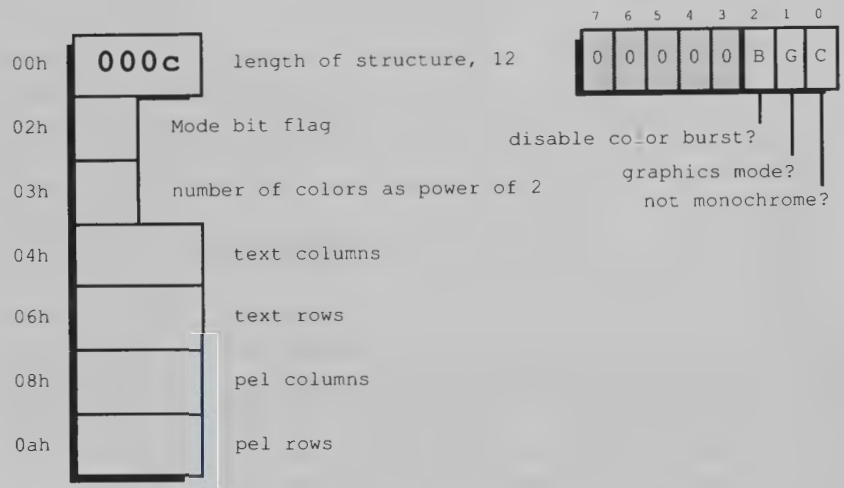
The configuration data could conceivably be wrong. It is based in part on hardware switch settings that might be wrong, and in part it represents assumptions that can't be checked, for instance the assumption that a video monitor is actually attached to the hardware and is appropriately configured.

However, there isn't much that a program can do in the way of an independent check on the configuration, and anyway if the display is misconfigured the system isn't going to process very many commands before crashing, so you might as well treat the configuration data as gospel.

10.4.1.2 The Display Mode

The *VioGetMode* call returns the current display mode in the form shown in Figure 10.2. The mode can be changed by getting this structure, altering one or more fields, and passing it to the *VioSetMode* call. This is a key data structure for display work; study it carefully.

Figure 10.2
Stucture Used in VioGetMode and VioSetMode



In MS-DOS systems, the display mode is represented by a single hexadecimal number, for example mode *1* or *7* or *F*. Each mode number stands for a particular way of interpreting the display buffer and for particular vertical and horizontal screen dimensions. However, mode numbers say too much *and* too little. Different display adapters implement the same numbered mode in different densities and the mode number doesn't distinguish between these. On the other hand, certain modes of the CGA are effectively identical and treated as such by the EGA and VGA.

OS/2 does not use mode numbers at all. Instead it uses the five numbers and three bits in the mode data structure. Every numbered mode can be represented there, and finer distinctions can be made as well.

10.4.1.2.1 The Mode and Colors Bytes

The *disable color burst* bit is used only by the CGA. That adapter has pairs of modes (old numbers 0, 1 and 2, 3) that differed only in whether the color burst signal was transmitted to the attached video monitor.

Monitors attached to the EGA and VGA do not use the color burst signal at all. Unless you have a specific objective in mind, the best way to handle this bit is to preserve it exactly as *VioGetMode* returns it.

The *color, not monochrome* bit is 0 when the MDA is installed. It will be 1 for any other adapter.

The graphics mode bit distinguishes all graphics modes from all text modes. When it is 0 the display buffer must contain character cells, when 1, pel bits. The remaining numbers in the structure must be interpreted in the light of the graphics-mode bit.

The number of colors field specifies how many simultaneous colors the screen can display, but specifies it in terms of the number of binary bits needed to encode those colors: one bit to encode a choice of two colors, white on black for example; two bits to encode either four colors or two colors with a choice of intensities; four bits to encode 16 colors. One of the VGA's graphics modes allows 256 colors, encoded in 8 bits.

10.4.1.2.2 Possible Modes

The text-mode dimensions are represented in two fields. When the display is in a text mode, these words (together with the *color not monochrome* and *disable color burst* bits) specify the hardware mode. The pel-dimension words that follow, merely document the screen resolution; they should be preserved as *VioGetMode* returns them. The possible text modes are shown in Table 10.3.

When the display is in a graphics mode, it is the text dimensions that are only documentation. The possible graphics modes are shown in Table 10.4.

10.4.1.2.3 Changing the Display Mode

VioSetMode is used to change the display mode. It takes the data structure of Figure 10.2 as input. The best way to set a new mode is to get the present mode, change only the necessary values in the data structure, and use that structure as input to *VioSetMode*.

If the fields of the structure contain a compatible combination of values, *VioSetMode* sets the mode (for the current screen group) instantly. The new interpretation of the display buffer goes into effect at once. If a graphics mode is begun while the buffer contains character cells, the screen displays a meaningless pattern of dots as a result of the character cell's bits being interpreted as pels.

When the mode is changed to a text mode, the character font may be changed. If it is, the cursor shape may be reset to an underscore form. In all cases the cursor position is reset to the upper left corner.

Table 10.3
Text Display Modes as set by VioSetMode

Color Exp.	Text cols.	Text rows	Pel cols.	Pel rows	Mode Number	Adapter Type	Char. Size
4	40	25	320	200	0, 1	CGA	8x8
4	80	25	640	200	2, 3	CGA	8x8
4	40	25	320	350	0, 1	EGA	8x14
4	80	25	640	350	2, 3	EGA	8x14
4	80	43	640	350	none	EGA	8x8
4	40	25	360	400	0, 1	VGA	9x16
4	80	25	720	400	2, 3	VGA	9x16
1	80	25	720	350	7	MDA	9x14
1	80	25	720	400	7	VGA	9x16

Only a mode supported by the physical hardware will be accepted by VioSetMode. The last two modes are monochrome.

Table 10.4
Graphics Modes as set by VioSetMode

Color Exp.	Text cols.	Text rows	Pixel cols.	Pixel rows	BIOS Mode	Adapter Type
2	40	25	320	200	4, 5	not MDA
1	80	25	640	200	6	not MDA
4	40	25	320	200	D	EGA or VGA
4	80	25	640	200	E	EGA or VGA
2	80	25	640	350	F	EGA or VGA
4	40	25	640	350	10	VGA only
1	80	30	640	480	11	VGA only
4	80	30	640	480	12	VGA only
8	40	25	320	200	13	VGA only

10.4.1.3 The Cursor Position

The *VioGetCurPos* call returns the present cursor position. The rows and columns are numbered from row zero and column zero at the upper left corner of the screen. This contrasts with the numbering scheme used in the ANSI escape sequences, which start numbering rows and columns with 1.

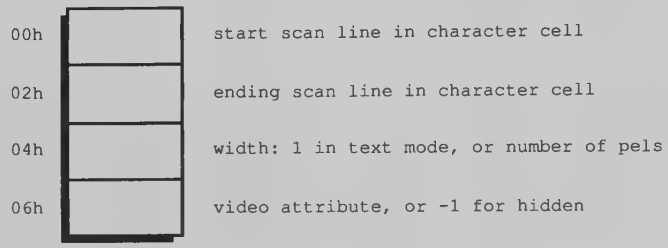
The *VioSetCurPos* call sets the cursor position to a specified row and column. The cursor position is set to 0, 0 by the *VioSetMode* call, so when setting both mode and cursor position, set the mode first and the position after.

10.4.1.4 The Cursor Shape

The cursor is generated by the hardware whenever a text mode is in effect. The cursor is a pattern of lines the width of one character box. It blinks on and off; when it is on it has the attribute (that is, the color and intensity) of a character symbol. When it is off it has the color of the background of that character box.

The *VioGetCurType* call returns information about the cursor in the data structure shown in Figure 10.3. The first two fields give the starting and ending lines of the cursor within a character box. These values depend on the height of the box, and this varies with the display mode. Box dimensions for each text mode are shown in Table 10.5.

Figure 10.3
Structure Used in VioGetCurType and VioSetCurType



The lines of a character box are numbered from zero (the top line) through its height less one (the bottom line, which is adjacent to the top line of the box on the next lower row). Typically the cursor's starting line is the next to last line of the box and its ending line is the last line. This results in a cursor that looks like a heavy underscore.

If the starting line is zero and the ending line is the last in the box, the cursor is a block the size of the box. If the starting line number is greater than the ending line, the cursor wraps around the box to make a double line. It starts low in the box and runs to the bottom, then reappears on the top line and continues down to the ending line.

The size of a character box, and hence the number of the bottom line, varies with the display mode and with the adapter that implements it (as can be seen in Table 10.3). There are two ways to find out the current box height. It can be inferred from the values returned by *VioGetMode*. Alternatively the *VioGetFont* call, discussed next, will return the box dimensions.

The cursor width field of the structure has little meaning in text mode. *VioGetCurType* returns it as zero. *VioSetCurType* will accept it as zero or as 1 (meaning a width of one character box).

The field billed as the cursor's attribute is not an attribute in the sense usually employed with text displays. Rather it is a flag that, when set to all-ones binary *(-1)* means that the cursor is turned off, invisible. When the cursor is visible, the field is zero.

The *VioSetCurType* call takes the data structure of Figure 10.3 and sets a new cursor shape for the current screen group. The shape is determined by the starting and ending line numbers as described, and the cursor will be made invisible or visible depending on the final field. The best way to change the cursor is to get its present shape, change the necessary fields, and use the structure as input to *VioSetCurtype*.

The shape of the cursor may be reset by a *VioSetMode* call, so when you plan to set both the mode and the cursor shape, change the mode first then the cursor.

10.4.1.5 The Current Font

The video font is the set of bit patterns that in text mode translates character byte values into displayed shapes. A font contains a bit pattern for each character value from zero to 255. The pattern specifies the shape that will be displayed in any character box that contains that character. The MDA has a fixed font. The fonts for other adapters are set by software.

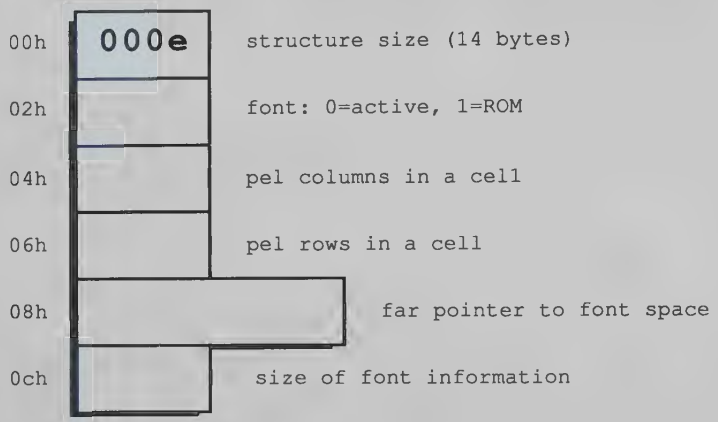
A font may also be called a *codepage*, in analogy to the codepage that is used to translate keyboard input. Indeed if a codepage is in effect for the keyboard, a related codepage should be in effect for the display. Otherwise the symbols being displayed might not all match the meaning of the keys being typed.

The font is first set when the system starts up. A default font may be set or, if the appropriate *devinfo* and *codepage* statements are given in the configuration file, a particular code page (font) may be loaded from a disk file. The *VioGetCP* and *VioSetCP* calls deal with fonts in terms of standard code page numbers, referring to entire font tables by single code page numbers.

The actual contents of the current font, and information about the font, are returned by the *VioGetFont* call. It fills in the data structure shown in Figure 10.4. Initialize the structure with its length in the first word and zero in all other fields.

Storage will be allocated for a copy of the font, and a pointer to it will be placed in the fifth field and its length will be set in the last field. (Alternatively the program may allocate its own storage to hold the font copy and provide the address.)

Figure 10.4
Structure Used in VioGetFont and VioSetFont



The size of a character box in the font is returned in the structure. (This information can be useful when changing the shape of the cursor.) The actual font data consists of 256 sequences of bytes, one sequence for each possible character. Each sequence contains one byte for each line in a character box, and specifies the display of the pels in that line of a box that contains that character. A 1-bit causes the pel to be displayed with the foreground attribute for the box; a 0-bit causes the pel to use the background attribute.

The *VioSetFont* call takes the same data structure as input and sets the current font. (This cannot be done for the MDA.)

The font information could be used in a number of ways. It could be used to guide the painting of character symbols after the display had been switched into a graphics mode at the same pel dimensions. It could be used as the basis for graphics output on a printer.

Alternatively, the font could be modified and set as a new font. For example, if the program needed a few special symbols not ordinarily present, it could get the current font, overwrite the bit maps of a few low-use character positions, and set the modified font with *VioSetFont*. Indeed, a program might replace many characters with graphics patches designed to provide a palette of shapes and textures for some particular image style—and thus equip itself to paint images with the speedy, convenient text-mode output calls.

References: ASCII code page 5.2.3; keyboard code page 11.2.3.1

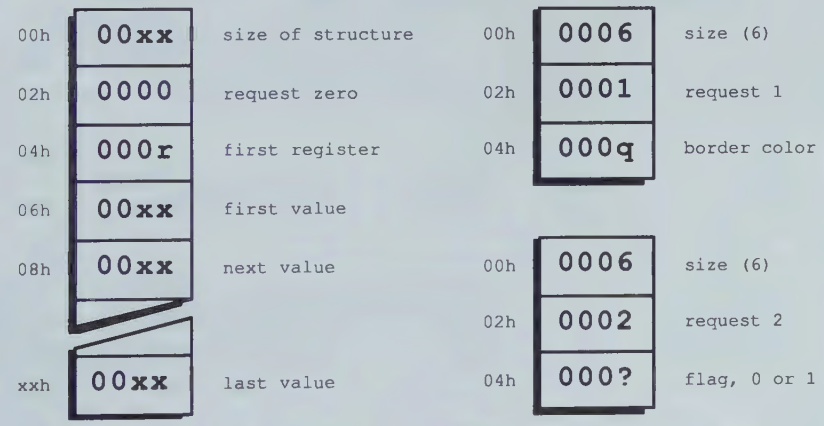
The font may be reset when a new text mode is set with *VioSetMode*, so perform font operations after setting the desired mode.

10.4.1.6 The Palette Registers

The EGA and VGA contain sixteen *palette registers* that determine the exact hues that will be displayed in text and graphics modes. These registers are set by the system while starting up and ordinarily aren't reset. However, they can be queried and set using the *VioGetState* and *VioSetState* calls.

These calls perform one of three miscellaneous services based on the data structure diagrammed in Figure 10.5. It is a rather unfortunate piece of software design that these calls perform such different services using such heterogeneous input structures; it makes the declaration of the data structure quite elaborate (a union in C, a variant record in Pascal).

Figure 10.5
Data Structures Used with VioGetState, VioSetState



At any rate, if the second word of the structure contains zero, *VioGetState* will return the contents of as many palette registers as the length of the structure permits, starting with the register indicated in the third word. Usually the length of the structure will be 38 (0026h), the first register will be register 0, and all 16 palette registers will be returned.

The returned values can be modified in place and the same structure handed to *VioSetState* to reprogram the palette registers.

10.4.1.7 The Border Color

In text modes (with any adapter except the MDA) the outer border of the screen has a color that may be different from the background colors of the character boxes. This color may be queried or set through *VioGetState* and *VioSetState* using the second format shown in Figure 10.5. As usual, it's best to employ the query call, modify the color value in place, and hand the structure back to *VioSetState* to change it.

10.4.1.8 The Attribute Bit 7 Switch

In text modes (with any adapter except the MDA) the most significant bit of a character attribute may have either of two distinct meanings. Under one interpretation, the bit means "blink this character box." Under the other, the bit means "high intensity background color."

The meaning may be queried and set with yet another use of *VioGetState* and *VioSetState*. The meaning of attribute bit 7 is returned in the third word of the structure shown in Figure 10.5. Zero is returned if the attribute bit has the usual (and power-on default) meaning of *blink*; 1 is returned if it is an intensity bit. The same code can be used to set the meaning.

10.4.2 Text Mode Operations

OS/2 contains a great deal of support for text-mode display output. Most of the calls discussed below are compatible with the Presentation Manager as well; under the Presentation Manager they operate within a default window and presentation space.

10.4.2.1 The Character Cell

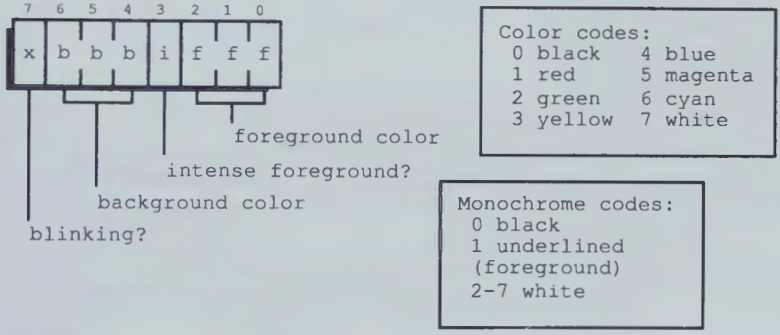
When the display is in any text mode, the hardware interprets the contents of the display buffer as character cells. Each cell is a 16-bit, or two-byte, field that specifies the contents of one character box on the screen.

The first byte of the pair contains a character code from zero to 255. Character number zero in all fonts is blank, so there are 255 nonblank symbols. The physical display buffer always begins on an even address (and the logical display buffer segment to be discussed shortly usually begins at an even offset) so the character byte is the byte at the even offset.

10.4.2.2 The Attribute Byte

The second byte of the pair is the attribute byte; it specifies the appearance of the character box on the screen. The contents of an attribute byte are shown in Figure 10.6.

Figure 10.6
Attribute Byte in a Character Cell




```
Bit 7 can mean either "blink" or "intense background color";
either meaning may be set with VioSetState
```

The basic meaning of an attribute byte is that the four least significant bits set the appearance of the character itself, while the four most significant bits set the appearance of the rectangular box within which the character is painted. For example, if the low-order bits are *0111b*, the attribute calls for a white letter of normal intensity. If the high-order bits are 0, the background of the letter is black. If these values are exchanged, the attribute calls for a black letter on a white background.

In a monochrome mode (the only mode afforded by the MDA) the only background color codes are zero for black and nonzero for nonblack. (Nonblack may be white, amber or green depending on the type of video monitor.) The foreground

color of a monochrome display may be zero for black, 1 for nonblack with an underscore, and any other value for nonblack without an underscore. The conventional usage is to set zero for black and seven for white, since this works correctly on a color monitor as well.

In any colored text mode, the color codes listed in Figure 10.6 apply.

Bit 4 of the attribute usually specifies high intensity for the foreground (character symbol) color. On a color monitor the brighter colors are distinct shades, quite different from the normal colors. In particular, intense color 5 is not bright magenta but yellow. (If you want to experiment with the foreground and background colors, note that all shades and intensities can be set using ANSI escape sequences. These are quite easy to program in a high-level language.)

It is possible to reprogram an EGA or VGA so that attribute bit 4, instead of selecting high intensity color, selects a different font of 256 symbols. That is not often done and there is no OS/2 system call for doing it. If you need to do it you will have to use an I/O privileged segment as described below for graphics modes.

Bit 7 of the attribute byte usually specifies that the foreground should blink; that is, that foreground pels should alternate between the foreground color and the background color. The alternation happens at the same rate, and in opposite phase, to the cursor's alternation between foreground and background colors.

It is possible to reprogram the display so that attribute bit 7 means *high intensity background color* instead *blink*. That can be done with *VioSetState*.

10.4.2.3 Text Output Operations

OS/2 provides a number of calls for writing data to a display in text mode. These are essentially string moves into the display buffer, but the program doesn't need to know the physical address of the buffer, and it specifies the starting location in terms of a row and column, not a buffer offset.

The simplest output call is *VioWrtNChar*. It writes one or more copies of a character number into the buffer. The output characters will use the attribute bytes that are already in place. This call can be used to set a single character at a designated row and column; alternatively it can be used to fill many cells with a replicated character. If the replicated character is the zero byte, this call can be used to implement the functions of clearing (to end of) the line or clearing (to end of) the screen.

A partner to *VioWrtNChar* is *VioWrtNAttr*; it writes one or more copies of an attribute byte into the buffer without affecting the character bytes that are already in place. This call can be used to change the attribute of a single character at a designated row or column; alternatively it can be used to change the appearance of many characters. It could be used to set the color, or to set a reverse-video effect (or blinking or high intensity) over a line or several lines. It might be used, for example, to change the color of a menu line when it was selected; or to set high intensity over the range of a wide, multi-character cursor area.

A third replicating call is *VioWrtNCell*; it writes one or more copies of a two-byte character cell into the buffer, replacing the cells that were there. This will most likely be used as an initialization call, to set up an area with a specified attribute.

The *VioWrtCharStr* call writes a specified sequence of character bytes into the buffer at a given row and column. It is the basic way of moving a string of characters into the buffer (using the attributes already in place). The *VioWrtCharStrAtt* call also writes a sequence of characters, but it takes a single attribute byte that is to be applied to every written character.

Finally, the *VioWrtCellStr* call writes a sequence of one or more complete character cells. This call is not slow; it is essentially a storage to storage move from program storage to the display buffer. It can be used to replace the entire screen contents in a flash. (However, for quick paging of whole screens see *VioShowBuff*, below.)

10.4.2.4 Text Input Operations

There are two calls that allow a program to retrieve the contents of the display buffer. The *VioReadCharStr* call retrieves a specified number of character bytes starting at a given row and column. It could be used to capture the contents of the display independent of its attributes, perhaps for printing or as a communications log. The *VioWrtCharStr* call can be used to write back bytes that *VioReadCharStr* saves.

The *VioReadCellStr* call retrieves the entire character cells, data and attribute both. This call is not slow; it is essentially a storage to storage move from the display buffer to program storage. It can be used to save the entire contents of the screen, and *VioWrtCellStr* can then be used to restore the contents.

10.4.2.5 Text Scrolling Operations

There are OS/2 calls that implement scrolling of arbitrary rectangular regions. They are called *VioScrollUp*, and *-Dn*, and *-Lf*, and *-Rt*, and they take identical parameter lists. Each takes

- the top row number of the region
- the left column number
- the bottom row number
- the right column number
- the number of blank rows or columns to be inserted
- the character cell to be used for fill.

All of them permit the bottom-row and right-column numbers to be given as minus-1, all-ones binary meaning "the end of the screen." Thus the whole screen can be scrolled by specifying row zero for top, column zero for left, and *-1*, all-ones binary, for bottom and right. This relieves the program of having to find out the current screen dimensions.

Most programs that use the scrolling calls will need to get the screen mode for other reasons. However, the scrolling calls, and in particular *VioScrollUp*, are intended as the standard way of clearing the screen. The call:

```
VioScrollUp(0,0,-1,-1,-1,BlankCell)
```

which supplies default values for bottom row, right column, and number of lines is the recommended way of clearing the screen. The *BlankCell* parameter would be a character cell containing either zero or ASCII blank for a character and the attribute byte desired for the whole screen.

The other uses of the scrolling calls are simply to operate rectangular scrolling regions of any size within the screen. The simplest vertical scroll would use *VioScrollUp* (or *-Dn*) to scroll an area by one line, filling the line with blanks. Then *VioWrtCharStr* could be used to write data bytes into the newly-cleared line. The simplest horizontal scroll would use *VioScrollRt* (or *-Lf*) to shift an area by one column, inserting a column of fill cells. Unfortunately there's no corresponding column-oriented write; data bytes would have to be put into the blanked column using a series of *VioWrtCharStr* calls (or with direct stores into the logical display buffer).

However, uses of scrolling are only limited by the programmer's imagination—

and, of course, by the limited images available in text mode, since the scrolling calls are not applicable to graphics modes.

The scrolling calls operate very quickly. Several sequential scrolls of one line are not perceptible to the eye as a scrolling motion, only as a blink. If you want the scrolling motion to be visible you will have to insert delays (*DosSleep* calls) between scroll motions.

The parameter lists of the scrolling calls are unpleasantly long, and the separate specification of row and column numbers makes it hard to store the shape of a region as a unit. But in fact, a scrolling *region* is a unitary object that is unlikely to change once defined. I recommend the creation of a region data structure to represent it. The structure would contain fields for:

- upper row and left column numbers
- lower row and right column numbers
- fill-cell value
- other application-oriented fields

Then region-oriented scroll subroutines become easy to define; as parameters they need only a region and a count of lines or columns to be scrolled. It also becomes easy to create a region clearing subroutine that takes only a single parameter.

10.4.3 The Logical Display Buffer

If the display output calls listed above are not sufficiently flexible, the next step is to gain addressability to the display buffer. Then the program may update character cells directly, and may contain routines for whatever kind of special display services it needs. This may be particularly important if the program was originally written for MS-DOS and contains logic based on treating the display buffer as an addressable array of cells.

The display buffer defined by the hardware is filled with the output of whichever screen group is in the foreground, and is reloaded when the operator switches to another screen group. However, there is a *logical display buffer* that is unique to each screen group. This buffer receives the display output produced by any program in the screen group, regardless of whether the screen group is in the foreground or not.

A program may gain access to the logical display buffer for its screen group by calling *VioGetBuf*. This call returns a 32-bit address to the logical buffer, and the buffer's length.

There is only one logical buffer per screen group, and access to it is shared between the system display routines and any programs that use *VioGetBuf*. (It is a segment shared by the method of *Giving*; see *DosGiveSeg*.)

Text-mode display output produced by any means and by any process in the screen group will be reflected in the logical display buffer. Modifications to the display buffer will eventually be reflected in the display. The system updates the physical display buffer from the logical buffer whenever the screen group is in the foreground and the system knows that a change has occurred in the buffer.

The key clause in the last paragraph was *and the system knows*. The system does not waste time copying the logical buffer into the physical one unless there are new data to display. It knows about display output produced by any *Vio* call, but it cannot know when a program has stored new data directly into the logical display buffer.

The program has to notify the system after it has changed the logical display buffer. It does this by calling *VioShowBuf*. That call takes an offset and a length for the updated span of data. It isn't essential to be accurate with these parameters—passing an offset of zero and the length of the buffer will work—but it is presumably more efficient to pass an offset and length that are a tight bound on the updated area.

The logical display buffer can be used for rapid paging. The program should allocate storage segments for as many display pages as it needs, and set up the display pages in them. To switch to a new page it can do a storage to storage copy into the logical display buffer (preferably using a single string-move instruction for speed) and issue the *VioShowBuf* call. Since this involves two storage to storage copies of the page data (one by the program and one by OS/2 into the physical display buffer) this is considerably slower than the hardware method of paging a display. However it is still faster than the eye can follow and the delay should not be perceptible to the operator.

The logical display buffer is not restricted to text-mode operations. If the display is in one of the simpler, CGA-compatible graphics modes, images may be built pel-by-pel in the logical display buffer and displayed by *VioShowBuf*.

Reference: *DosGivSeg* 7.4.3

10.4.4 Pop-Up Displays

A *pop-up* display is a text-mode display screen that preempts whatever the operator was looking at. The *VioPopUp* call creates the display and seizes the screen to show it, forcing whatever screen was active into the background temporarily. The *VioEndPopUp* call terminates the display and restores the preempted screen group to visibility.

A pop-up display can be done by any process: the one in the foreground, one that is running in the background, or one that is running detached from any screen group. Because of its peremptory nature, and because it overrides the operator's choice of screen groups, a pop-up should be used only to display information that urgently requires the operator's attention.

A pop-up display is really an entire screen group; that is, a virtual screen, keyboard, and mouse. The process that issues the *VioPopUp* call may use system calls for keyboard, mouse and display functions while interacting with the operator. However, there is only one pop-up screen group, and only one process may use it at any time.

Since the pop-up screen group might be in use when a program requests it, the *VioPopUp* call can be asked to suspend the process until the pop-up screen group is available. The *VioEndPopUp* call releases the pop-up screen group entirely, and reconnects the previous screen group to the display, keyboard, and mouse.

VioPopUp initializes the pop-up display to an 80x25 text mode and clears the screen to ASCII blanks with background attribute zero and foreground attribute seven (the usual attributes of white letters on black background, regardless of the type of display adapter). The cursor is initialized to the usual underscore form and located in the upper left corner.

Only a subset of the display functions are available within a pop-up (see Table 10.4). Besides *VioEndPopUp* there are only the scrolling functions, the cursor functions, and the functions to write characters and cells. *VioWrtTTY* is supported, so stream output in general is available, including support for *DosWrite* and for ANSI escape sequences.

Table 10.4
Usability of Vio Functions in Different Environments

Function Name	During Pop-Up	Family A.P.I.	Presentation Manager
VioDeRegister	no	no	no
VioEndPopUp	yes	no	yes
VioGetANSI	no	no	yes
VioGetBuf	no	yes	yes
VioGetConfig	no[1]	yes	yes[2]
VioGetCP	no[1]	yes	yes
VioGetCurPos	yes	yes	yes
VioGetCurType	yes	yes	yes
VioGetFont	no[1]	yes	no
VioGetMode	yes	yes	yes
VioGetPhysBuf	no	yes	no
VioGetState	no[1]	yes	no
VioModeUndo	no	no	no
VioModeWait	no	no	no
VioPopUp	no	no	yes
VioPrtSc	no	yes	yes
VioPrtScToggle	no	yes	yes
VioReadCellStr	no[1]	yes	yes
VioReadCharStr	no[1]	yes	yes
VioRegister	no	no	no
VioSavRedrawUndo	no	no	no
VioSavRedrawWait	no	no	no
VioScrLock	no	yes[3]	no
VioScrollDn	yes	yes	yes
VioScrollLf	yes	yes	yes
VioScrollRt	yes	yes	yes
VioScrollUp	yes	yes	yes
VioScrUnlock	no	yes	no
VioSetAnsi	no	no	yes
VioSetCP	no	no	yes
VioSetCurPos	yes	yes	yes
VioSetCurType	no[1]	yes	yes
VioSetFont	no	yes	no
VioSetMode	no	yes	yes[4]
VioSetState	no	yes	no
VioShowBuf	no	yes	yes
VioWrtCellStr	yes	yes	yes
VioWrtCharStr	yes	yes	yes
VioWrtCharStrAtt	yes	yes	yes
VioWrtNAttr	yes	yes	yes
VioWrtNCell	yes	yes	yes
VioWrtNChar	yes	yes	yes
VioWrtTTY	yes	yes	yes

1 Officially not allowed, but works in release 1.0
2 Adapter RAM size not returned
3 Null operation, returns success immediately
4 Only field respected is number of text rows

10.5 Graphics Mode Operations

In contrast to text-mode displays, OS/2 contains little support for the graphics modes. In fact, it is almost unaware of the graphics modes and needs the application's help to save and restore the display state when switching in and out of a screen group that's in a graphics mode. There is no graphics parallel to the extensive list of *VioWrt* and *VioScroll* calls that support text mode. There isn't even the crude support of the old BIOS functions for reading and writing one pel by row and column position.

Many of the operations that follow are not permitted by the OS/2 Presentation Manager. A program that uses them cannot run in the Presentation Manager's screen group. (See table 10.4 for a list of which functions may be used under the Presentation Manager.) The discussion here is directed to programs that are called from the OS/2 command shell.

10.5.1 Saving the Text-Mode State

It's a good idea for a graphics program to save the state of the text-mode screen on entry and to restore it on exit. If the display is to be restored, what data shall it display? It could be restored to a clear screen, but it's much more impressive to restore exactly the screen that existed when the program began, prior commands and all, and it's not difficult to do this.

Here are the operations to be done. First use *VioGetMode* to get the display mode information. If the palette registers are going to be reprogrammed, use *VioGetState* request 0 to get their present contents. Use *VioGetCurType* to save the cursor shape, and *VioGetCurPos* to save its row and column position. Multiply the text row and column dimensions from the mode structure to get the display size in character cells. Allocate a buffer of twice that many bytes and use *VioReadCellStr* to copy the entire current display into it.

The display state is now safe and the program may change the display any way needed. In particular it may use *VioSetMode* to set a graphics mode.

When the program is finished, it should restore the display in the following sequence. If the palette registers were reprogrammed, restore them with *VioSetState* request 0. Restore the mode with *VioSetMode*. Use *VioWrtCellStr* to write back the display contents that were saved. Use *VioSetCurType* and *VioSetCurPos* to restore the cursor.

The effect is quite impressive: the operator hits the key or types the command that terminates program, there's a flash, and the screen is restored to what it was an hour before.

What if the program's end is unplanned? What if it divides by zero or is cancelled by control-break? A program that is meant for general use should cover these possibilities by setting up an Exit routine with *DosExitList*. The logic to restore the screen should be in the Exit routine.

Messages from OS/2 explaining the abnormal termination will be written to the screen after the Exit routine finishes, so they'll appear on the restored screen after the command line that started the program. It's very impressive when a dying program can bury itself so tidily.

References: getting and setting the display state 10.4.1; saving and restoring the buffer 10.4.2.4 setting up an Exit list 5.3.3

10.5.2 The Physical Display Buffer

In order to create a graphics image, the program must set and reset bits that represent pels. In the graphics modes compatible with the CGA, it can operate in the logical display buffer, but for higher modes it must write into the physical display buffer. However, it would be catastrophic if the program did that when the display buffer held the display image for a different screen group. Two things are needed: a way to address the buffer, and some way of knowing when it is safe to use the address.

10.5.2.1 Getting A Buffer Address

The *VioGetPhysBuf* call uses the data structure shown in Figure 10.7. It will insert segment descriptors in the program's address space to address the physical display buffer. It requires a starting physical address, that is, a simple 32-bit integer specifying the first buffer byte that the program wants to address (not a pair of 16-bit segment and offset words) and the length of the buffer wanted. It returns the one or two selectors needed to address the buffer for the size requested. The starting addresses and lengths of the buffers for different modes are shown in Table 10.5.

Figure 10.7
Structure Used in VioGetPhysBuf

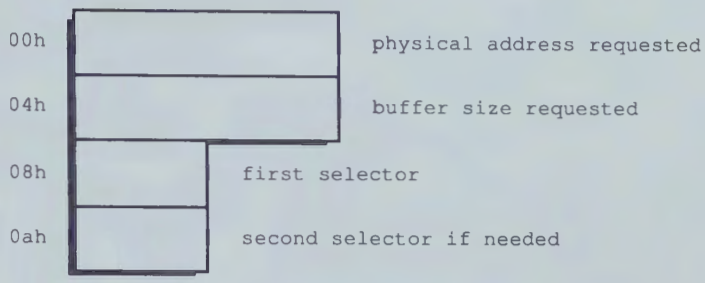


Table 10.5
Hardware Features of Graphics Modes

BIOS Mode	Buffer Start	Buffer Length	Number Pages	Page Offset	Comment
4, 5	b8000h	bf40h	1	n.a.	Even lines at +0000h, odd at +2000h
6	b8000h	bf40h	1	n.a.	Even lines at +0000h, odd at +2000h
D	a0000h	ff40h	8	2000h	Four planes at same address
E	a0000h	fe80h	4	4000h	Four planes at same address
F	a0000h	ed60h	2	8000h	Two planes at same address
10	a0000h	ed60h	2	8000h	Four planes at same address
11	a0000h	9600h	1	n.a.	
12	a0000h	9600h	1	n.a.	Four planes at same address
13	a0000h	fa00h	1	n.a.	one byte per pixel

Graphics modes are set with VioSetMode as shown in Table 10.4. The Page Offset is the byte distance from the start of one page to the start of the next one.

10.5.2.2 Locking the Buffer

The *VioScrLock* call makes the physical buffer safe to use. This call enables a program to suspend itself until its screen group is in the foreground, and then to lock its screen into the foreground so that the operator can't switch away from it.

Once it has the screen locked to the physical buffer, the program should do its work in the buffer without delay and then issue *VioScrUnLock*. Unlocking the buffer doesn't cause a screen switch (on the contrary, the operator will no doubt want to sit and admire the new image for some time), it only makes a screen switch possible—makes it possible, for example, for a hard error pop-up display to be displayed, something that can't happen while the physical buffer is locked.

The phrase "without delay" has a specific meaning in this context: while the screen group is locked, the program ought to avoid anything that could cause an indefinite delay. It should not do any file I/O, or wait on a semaphore, or wait for operator input of any kind. Processing may take significant time (especially if floating-point calculations are needed); that's all right. What the program mustn't do is to get itself suspended waiting for some external event.

10.5.2.3 Guarding Against Swap Delays

The danger is that, while the program has the buffer locked, the system could find a need to switch away to another screen group. There are two possible causes: the operator could be hammering on the hot key; or another process could request a pop-up display. The second possibility could arise, for example, if another program encountered an I/O error. If this happens, what is the system to do? It needs to use the display buffer, but the buffer is locked.

The system's policy is to wait an unspecified (but no doubt small) number of seconds. If the program hasn't called *VioScrUnlock*, OS/2 will switch away regardless. After all, the program might be in a hard loop!

When making such a forced screen switch, OS/2 will freeze the offending process, making all of its threads ineligible to run. That ensures that it won't continue to modify the display buffer while OS/2 is setting up a different display. At some unpredictable later time the system will switch back to the original screen group, and the program will be thawed out and allowed to continue building its display. However, it is not at all clear what the state of the display hardware will be at that time. It's likely that some of the work the program did before being frozen will have been spoiled.

There is one delay that the program can't really control: the unpredictable wait for swapped-out segments of code or data to be swapped back in. Normal swapping delays shouldn't be long enough to trigger a forced screen switch, but they can be avoided entirely in this case.

Here is a good sequence for operating in the physical buffer. First, assemble the image data in an ordinary storage segment. Whatever calculations and file I/O that are needed may be done without fear of delaying the system (they can, in fact, be done while the program's screen group is in the background).

When there is nothing more to do except display the image, use *VioScrLock* to suspend the program until its screen group is switched to the foreground. Of course, the screen group may already be in the foreground; if all this work was triggered by operator input, it must be in the foreground or it couldn't have received the keyboard or mouse signal. In that case, *VioScrLock* will take almost no time to execute.

Nevertheless the program ought to immediately issue *VioScrUnlock*! Now that it is sure of being in the foreground, it should make sure of its segments by touching (that is, referencing data in) each segment it will need while updating the screen. There should only be a few: the segment into which the display procedures have been linked, perhaps an I/O Privilege segment, and the segment(s) of image data. Finally the program may gain control of the buffer with another *VioScrLock* call (which will almost certainly return at once), copy the image to the physical buffer, set the adapter registers, and call *VioScrUnlock*.

This procedure might seem ridiculously complicated, but it can't do any harm, and it adds very little overhead, and it averts the catastrophe that would follow, if, after it had locked the buffer, it should access a swapped-out segment and there should be an I/O error during the swap-in.

References: linking IOPL segments 3.3.4.5.4, 3.3.4.7

10.5.3 Programming the EGA Registers

The older graphics modes of the CGA, and the 256-color mode of the VGA, can be operated with no input other than the palette registers and the bits stored in the display buffer. The enhanced modes of the EGA aren't so simple. These modes place multiple bit planes in overlapping storage addresses. Consider the 16-color, 600x320 mode for instance (see Table 10.4). Here each pel is defined by four bits. The byte at a0000h, for instance, contains the first of four bits from each of

the first eight pels—and the second of four bits, and the third, and the fourth bits of the same pels, but only one bit of each pel at any time. Registers in the EGA must be set to define which of four bit planes is to be addressed when the buffer is updated. It's also possible to program the hardware so that it will combine bits from different planes, or store identical bits into multiple planes.

EGA programming is beyond the scope of this book. Here arises the question, how can a program operate on the hardware registers? There are no OS/2 system calls for doing this. The answer is that the program must contain the I/O instructions to read and write the registers. This requires I/O Privilege. It is possible to define a segment of code as needing I/O privilege when the segment is linked. When the segment is loaded as part of the program, it is tagged for I/O privilege, and while the program is executing in that segment it may execute *IN*s and *OUT*s.

The best way to design an IOPL segment is as a set of small subroutines with specific functions. This will minimize both the size of the IOPL code and the trouble of debugging it. That's good, since it will probably have to be written in assembly language.

10.5.4 The Save-and-Redraw Watchdog

OS/2 knows how to save and restore the display state of a text-mode application, but not that of a graphics application. It doesn't know how to save the larger display buffer used in graphics modes. In particular it doesn't know how to save the many register values that a program may set in the EGA or VGA control registers. Some of these are write-only registers in any case, so there's no practical way for the system code to capture their values.

(Parenthetically let us note that it would have been feasible, though difficult, to virtualize the EGA registers on a screen-group basis. It could have been done using a concept like the shadow registers introduced in the firmware of the Atari 800, to name one possible model. However, it was not done, and that's that.)

When the operator wants to switch to another screen group, who will save the buffer contents? And when the operator switches back, who'll repaint the screen? The program has to provide the code to do these things. It could have presented a sticky design problem, since the operator's actions are unpredictable, asynchronous events. The problem was simplified with a rather elegant application of OS/2 multiprogramming.

The idea is that a graphics program should create a separate, asynchronous thread to guard the display like a watchdog. The thread spends most of its time sleeping. When it's time to save or restore the state of the display, the system wakens the sleeping code; it does the work and goes back to sleep.

The *VioSavReDrawWait* call is the means by which the watchdog sleeps until it is needed. The system suspends the caller until it is time to switch into or away from the caller's screen group. Then the call ends and control comes back to the watchdog code, which knows that something is about to happen. A signal is returned indicating which event is about to occur:

- The system is about to switch into this screen group; the hardware state and buffer contents should be reconstructed
- The system is about to switch to another screen group; the hardware state and buffer contents should be preserved.

In many applications the image data and the necessary state of the graphics registers can be recreated entirely from program data; no saving is needed. In that case the watchdog can arrange to be awakened only for the redraw event. Restoration of the hardware state after a pop-up display is handled by a separate *ModeWait* watchdog, discussed below.

The plan is that the graphics application will create a separate thread to sit forever in this watchdog loop:

```
repeat
  rc = VioSavRedrawWait(what)
  if rc is zero then
     if what=save then
        save the graphics state
     if what=redraw then
        reset the state and repaint the picture
until rc isn't zero.
DosExit
```

Careful design is needed where the save/redraw thread and the main thread interact. The need to switch between screens is an asynchronous event; it can happen at any time that the screen isn't locked with *VioScrLock*. The design will probably be simpler if saving isn't necessary, that is, if redrawing can be done entirely from program data. Then the watchdog thread can treat the main thread's variables as read-only data and there is no need to synchronize access to the shared items.

The need to save the EGA or VGA registers can be eliminated if the main-line code that writes to them always writes identical values into shadow registers in storage. Then the redraw thread can perform a complete restoration of the hardware state very efficiently by copying the shadow registers into the real ones.

Part of orderly termination of a graphics application is to cancel its watchdog. That's done with the *VioSavRedrawUndo* call, which will either terminate the thread or cause control to return to it with a nonzero return code.

Reference: creating threads 9.1

10.5.5 The Mode-Restore Watchdog

When the system displays a pop-up it apparently doesn't do as thorough a job of destroying the display state as when it switches in a normal screen group. We may infer that it saves and restores the contents of that part of the display buffer that it uses, so there is no need to repaint the image.

What OS/2 can't do after showing a pop-up is to restore the setting of the EGA or VGA registers, particularly the mode and palette contents. A graphics application is expected to create a separate thread, a second watchdog similar to the save/redraw thread, to restore the adapter mode.

The *VioModeWait* call is the way this watchdog sleeps. It is expected to sit in a loop like this one:

```
do forever
   rc = VioModeWait
   if rc isn't zero, leave loop
   restore the adapter state
end loop
DosExit
```

The watchdog is only called to restore the hardware state, not to save it. There is no notification when a pop-up is being started. This makes it all the more important to have the code that alters hardware registers also maintain "shadow registers" in storage. The main thread (or the Exit procedure of the program) may use *VioModeUndo* to cause the *VioModeWait* call to end with a nonzero return code. That will cause the watchdog to wake up and return to its kennel.

References: creating threads 9.1; pop-up displays 10.4.4

10.6 Replacing Display Functions

Most of the display system calls can be replaced with your own code. The replacement code takes over the functions of the replaced calls for the entire screen group. The calls that can be replaced are shown in Table 10.6. Ordinarily this mechanism would be used by a program that creates and manages the execution environment for other programs, a shell like the OS/2 Presentation Manager.

The process is called *registering* a replacement. It is done via *VioRegister* and rescinded with *VioDeRegister*. One or more replacement functions may be registered at a time, but only one registration call is allowed to be in effect at a time within a screen group.

The code for all replaced calls is contained in a single procedure. Its entry point is called whenever any of the replaced functions are called by code anywhere in the screen group. When it is entered, the stack is arranged as shown in Figure 10.8.

The client program, the one that called the display function, has stacked parameters for it. Below these on the stack is the client's return address, then some words stacked by the display router, the system code that intercepts display function calls and routes them to replacements. One of these words is the index number of the replaced function as given in Table 10.6.

Usually, a replacement display function will be invoked when a process in the screen group makes that call. This will be true whether the screen group is visible or in the background. On the other hand, processes in other screen groups are unaffected; they go on using the standard display calls or, possibly, the replacements that are registered in their screen groups.

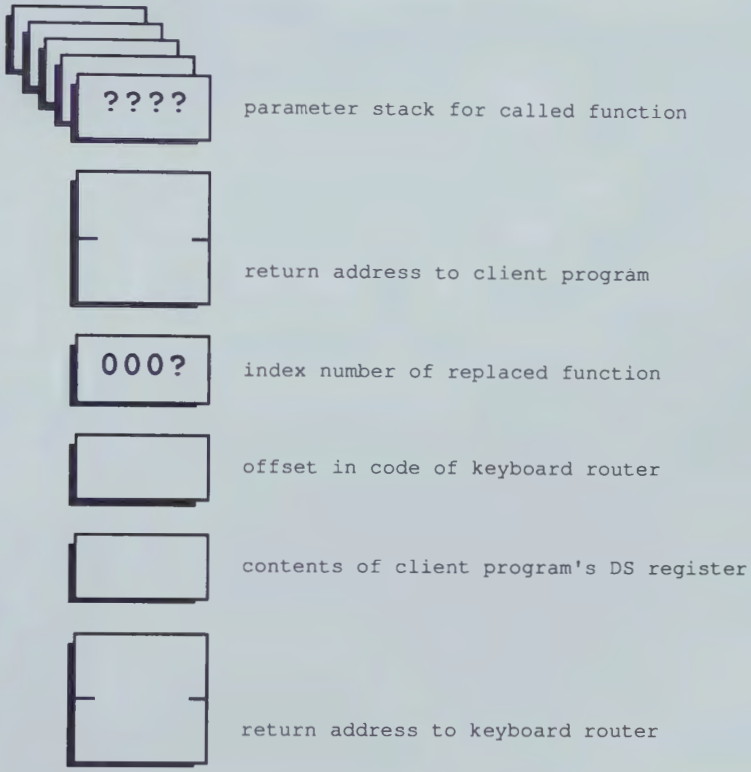
The replacement for *VioPopup* operates under different rules. It will be invoked whenever a pop-up display is requested while this screen group is visible. The invoking process may be in another screen group or may be detached from any screen group; the replacement *VioPopup* is invoked because the registered procedure is presumably the only one that knows how to implement a pop-up while its display is active. On the other hand, the replacement *VioPopup* will *not* be invoked, from its own screen group or any other, when its screen is not visible.

Table 10.6
Index Numbers of Replaced Display Functions

Number	Function
0	VioGetCurPos
1	VioGetCurType
2	VioGetMode
3	VioGetBuf
4	VioGetPhysBuf
5	VioSetCurPos
6	VioSetCurType
7	VioSetMode
8	VioShowBuf
9	VioReadCharStr
10	VioReadCellStr
11	VioWrtNChar
12	VioWrtNAttr
13	VioWrtNCell
14	VioWrtTTY
15	VioWrtCharStr
16	VioWrtCharStrAtt
17	VioWrtCellStr
18	VioScrollUp
19	VioScrollDn
20	VioScrollLf
21	VioScrollRt
22	VioSetANSI
23	VioGetANSI
24	VioPrtSc
25	VioScrLock
26	VioScrUnlock
27	VioSavRedrawWait
28	VioSavRedrawUndo
29	VioPopUp
30	VioEndPopUp
31	VioPrtScToggle

The index number of a replaced function is its bit number in the registration bit mask, and a word parameter to the replacing procedure code.

Figure 10.8
Stack Layout Entering Replacement Display Function



There are a number of restrictions on the code that is registered as a replacement. First, since it may be entered from more than one program, it must be located in a dynamic link library. Only dynalink code segments can appear in the address space of more than one program.

Second, the code must be *reentrant*, that is, it must permit concurrent execution by more than one thread (this follows from its being available to more than one program). If it uses a segment of static data it must serialize access to it with semaphores (not critical sections, as these are only effective within a single process). The simplest scheme is to define a semaphore among the static data items and to issue *DosSemRequest* for it on entry to the single replacement procedure.

Finally, the code cannot use the replaced display functions itself and so must carry out their operations by calling the video device driver with *DevIOCtl*, and/or performing direct hardware operations from an I/O privileged segment. Because of this, and because of the variable stack layout and the need for reentrancy, the replacement function will probably be written in assembly language.

Reference: semaphores as keys 9.4.2

11

Keyboard Input

Most applications can get by without direct access to keyboard input. After all, a program begins with a standard input file. If that isn't opened to the keyboard, it's because the operator redirected it to some other source of input, and a program should have a very good reason to ignore the operator's intent.

Then too, keyboard input can be had using the file interface, using *DosOpen* to open a file to the name of the keyboard device, *KBD$*, and using *DosRead* to get input.

The disadvantage to both these methods, of course, is that they produce only ASCII input. They never return function keys, arrow keys, or those exotic signals like control-alt-slash. There are applications that need this wider channel for operator control, and direct keyboard input provides it.

References: opening device names 8.7.1.1; *DosRead* 8.11.2

11.1 Is a Keyboard Available?

Before a program tries to use the keyboard, it should test to see if it has a keyboard available to it. A program started with a Run command in the configuration file is not part of a screen group, and therefore has no keyboard access. The same is true of a program started by the Detach command or by *DosExecPgm* with option 4, and of any program that is a child process to either of these kinds of programs.

How can a program tell that it has no screen group? There are two fairly simple tests. The first is to open a file to *KBD$* and then to use *DosQHandType* to get

the device driver attribute word of the file. If bit zero is not on in this word, signifying the standard input device, the program has no keyboard access. (The device word probably has bit 2, null device, set.)

The other test is to use the *KbdPeek* call to sample the keyboard; if it returns any nonzero code the program probably has no keyboard access. (Code 464 explicitly means "detached process has no keyboard access")

If the program has no access to a keyboard, it should issue a rueful message to the standard error file and terminate.

References: *DosExecPgm* 6.1.2; *DosQHandType* 8.10.2

11.2 Keyboard Decoding

The keyboard of an OS/2 system is an independent hardware device with its own CPU. It has a two-way data channel to the computer. Whenever any key is pressed, the keyboard notifies the computer by interrupting it and presenting a key *make* signal. When any key is released the keyboard again interrupts the computer and presents a *break* signal. The keyboard device driver interprets this stream of signals.

11.2.1 Scan Codes

The make and break signals are in the form of bytes called *scan codes*. Scan codes are simply arbitrary numbers for the key positions. The raw scan codes represent keys based on their position on the original IBM PC, plus codes for keys added to later keyboards. The OS/2 keyboard device driver processes the raw scan codes using a complicated set of rules for the intepretation of alt- and control-shift states. After processing, the set of scan codes a program will receive is shown in Table 11.1 ordered by key and shift state, and repeated in Table 11.2 in numerical order. (These tables have been placed at the end of this chapter owing to their bulk: see pages 344 and 347.)

Table 11.2 includes all scan codes, including some like *shift* and *NumLock* that are normally not returned on input.

11.2.2 Shift States

The keyboard hardware doesn't understand shifted keys; it treats the left and right shift keys, the left and right alt and control keys, and the three Lock keys just like the letter keys, merely notifying the computer of makes and breaks. The OS/2 keyboard device driver decides whether the present state of these keys is such as to put the keyboard into a certain shift state, or not.

It is also the device driver that decides when one of the keyboard LEDs should be lit, and tells the keyboard to light it. (You have probably noticed how OS/2 keeps the state of the keyboard LEDs on a screen-group basis, and how the LEDs will change as you switch from screen to screen. This is done by the session manager, which instructs the keyboard device driver to change the shift state to that of the incoming screen group. The device driver commands the keyboard to change the LEDs.)

11.2.3 National-Language Keyboards

The keyboard produces a stream of make and break scan codes, and it's up to the keyboard device driver to convert these into a stream of data that includes uppercase and lowercase ASCII, control characters, and extended codes for the myriad shifted and unshifted function keys. The decoding job is complicated by the use of different keyboard layouts in different nations. The keyboard hardware doesn't change from country to country, nor do the scan codes. However, the symbols on some of the keycaps do differ, and therefore so must the interpretation of some of the scan codes. Some languages use accented characters that take more than one keystroke to enter, and some have characters that are delivered in two bytes.

11.2.3.1 Keyboard Code Pages

The changed interpretation is brought about by changing the *code page* used by the device driver. The code page (for the keyboard) is a table that tells the device driver the meaning of each of the 127 *make* scan codes that the keyboard can deliver, with respect to each of the shift states (unshifted, shifted, control and alt).

The code page can also designate some keys as *accent* keys. These keystrokes, when unshifted, are taken as a modifier on the keystroke that follows. The code page also designates which scan codes may be accented. If an accent keystroke is followed by an accentable keystroke, the pair is combined into a single, accented,

character in the extended ASCII set. An accent key with control or alt is taken to be an independent symbol, and an accent key modified by the same accent is also taken as that accent symbol. An accent key followed by a key that can't be accented is also taken as two independent symbols.

The code pages that the keyboard may use are established at bootstrap time by statements in the system configuration file. The statement that begins "devinfo kbd$,..." is used to inform the keyboard device driver of the layout of the physical keyboard (the second parameter) and to load a file of code pages (the third parameter of the statement). If this statement is omitted in the configuration file, codepage switching is not enabled for the keyboard.

A later statement of the form *codepage=n* is passed to all device drivers that support codepages, and asks them to initialize with the given code page number. If this statement is omitted from the configuration file, the keyboard is initialized to a default code page.

11.2.3.2 Double-Byte Character Sets

The alphabets of some nations have more symbols than can be represented in a single byte. These alphabets are represented in two-byte codes, and such an alphabet is a Double-Byte Character Set, or DBCS. When the keyboard device driver is translating into a DBCS, some returned characters are two bytes long.

The *DosGetDBCSEv* call returns a list of byte values that may be the initial bytes of double-byte characters. Ordinarily the list is null. When it is not, every ASCII code received from the keyboard must be checked. If it is a DBCS initial byte, the second byte must be read to complete the character.

11.3 Keyboard Sharing

OS/2 has to apportion the one real keyboard among all the programs that are running in the system. It has two methods of doing this: screen groups and logical keyboards.

Each screen group contains a one virtual keyboard. When a screen group is in the foreground, that is, visible on the screen, its keyboard is identified with the real keyboard hardware. When the screen group is in the background, that is, not visible, its virtual keyboard is disconnected from the hardware. The operator's

keystrokes are directed to the foreground keyboard. The ones in the background ones see no hardware activity.

This would be a sufficient sharing mechanism if there was a limit of one program per screen group, but that is not the case; a screen group may be home to many concurrent programs. For example, the operator can start any set of programs merely by naming them in a command line with ampersands between them.

Most programs need only the standard input file and do not require direct keyboard input. But there is still a chance that two or more programs might both want direct use of the same virtual keyboard. There is a method of dividing the virtual keyboard of the screen group into separate *logical keyboards*, and a way of sharing keyboard input among them called the *keyboard focus*.

11.3.1 Logical Keyboards and Keyboard Focus

The *KbdOpen* call creates a logical keyboard. It returns a handle that stands for that logical keyboard. Initially the keyboard handle can't be used because it does not have the *keyboard focus*. The keyboard focus is simply the exclusive right to the keystrokes arriving at the virtual keyboard.

The keyboard focus is obtained with the *KbdGetFocus* call. It takes a handle created by *KbdOpen* and makes the connection between the logical keyboard and the virtual keyboard of the screen group. Once the logical keyboard has the keyboard focus, no other keyboard created by *KbdOpen* may have it. The sequence of *KbdOpen* followed by *KbdGetFocus* is intended to give a program exclusive ownership of the virtual keyboard of its screen group—and it will, provided that all other programs use *KbdOpen* as well.

Once it has the keyboard focus, the logical keyboard has all the attributes of the real keyboard: it has a code page (which may be different from the code page of other logical keyboards) and a shift state (which will be reflected in the keyboard LED indicators) and an input buffer.

There is a *KbdFreeFocus* call with which a program may give up the connection between logical and virtual keyboard while still retaining the handle and its attributes for later use. The focus is gotten back with another *KbdGetFocus* call.

When the program is finished with the logical keyboard it may dispose of the handle with a *KbdClose* call. Closing a handle gives up the focus as well. A keyboard handle will be closed automatically as the process ends.

11.3.2 The Default Logical Keyboard

Most of the *Kbd* series of system calls take a keyboard handle as a parameter, so that they can be applied to a logical keyboard created by *KbdOpen*. However, it is not necessary to create a logical keyboard to work with the keyboard.

There is a *default* logical keyboard that is represented by a handle of zero. This is the keyboard whose buffer receives input when no other keyboard has the focus. It does not hold the focus, in the sense that it is not necessary to call *KbdFreeFocus(0)* in order to use *KbdGetFocus* on a nonzero handle.

A program may choose to ignore the whole mechanism of logical keyboards and keyboard focus, and simply use the default keyboard, supplying a handle of zero wherever a keyboard handle is required. What, then, is the point of the logical keyboard mechanism? What, indeed! If *all* programs that work directly with the keyboard use *KbdOpen* and *KbdGetFocus*, the sharing mechanism will work. But if *any* programs use the default keyboard handle, sooner or later there will be contention between two programs for the use of one virtual keyboard. The results will be unpredictable, except that we can reliably predict a confused operator.

The preferred and recommended way of using the keyboard is to coin a handle with *KbdOpen* and to get the focus with *KbdGetFocus*. This only needs to be done once at the start of a program and costs next to no time. Then if a problem should arise from keyboard contention, you may confidently point a finger at the other programmer.

11.3.3 The Keyboard Input Buffer

Each logical keyboard has an associated input buffer. So long as a keyboard has the focus, any incoming keystroke will accumulate in its buffer until a program reads it. This permits the operator to type ahead of the program, and prevents the overruns and loss of data that might occur if the program was slow in processing keystrokes for a fast typist.

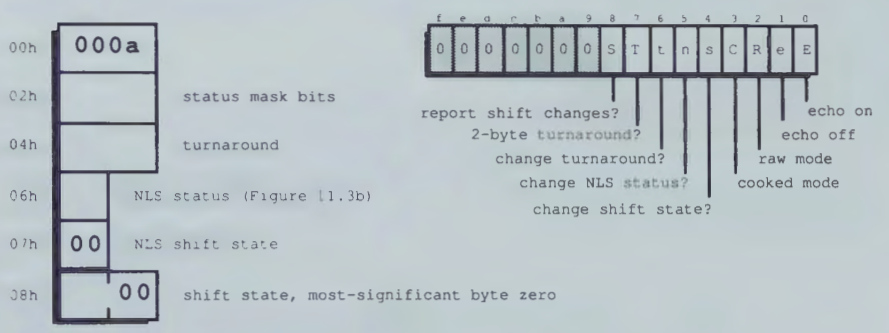
Sometimes buffered input is a liability. If the operator is going to be asked to signal a critical choice in a single keystroke (as in "Destroy all files (y/n)?") it is a good idea to *not* accept any buffered input. It's important to be sure that the operator's finger is responding explicitly to the current question.

]The *KbdFlushBuffer* call permits this. It empties the input buffer so that the next input data is sure to have been entered later than the time the call was made.

11.4 The Keyboard Status and Modes

The *KbdGetStatus* and *KbdSetStatus* are complementary calls for checking or changing the state of a logical keyboard. They operate on a data structure that holds a miscellany of state information (see Figure 11.1).

Figure 11.1
Keyboard Status Structure Used in KbdGetStatus and KbdSetStatus



KbdGetStatus fills the structure with the current status. *KbdSetStatus* sets the keyboard from the contents of the structure. Two important modes of keyboard input are reflected in the mask bits in the second word: Echo mode and raw versus cooked mode.

11.4.1 Echo Mode

When echo mode is on, input keystrokes are echoed to the screen; when it's off, they aren't. In MS-DOS there were different system functions for echoed and unechoed input. In OS/2 there are a variety of ways of reading from the keyboard (*DosRead*, *KbdCharIn*, *KbdStringIn*), and all of them echo or do not echo according to the echo mode in effect.

Echo mode is on by default. Whenever the screen output isn't to be a simple echo of the input, the program may turn echo mode off and give the operator whatever visual feedback the input keystrokes require by some other means.

11.4.2 Raw and Cooked Input Modes

The keyboard device driver processes input in one of two modes whimsically called raw and cooked. The difference is largely in which keystrokes the device driver will interpret as a special signal, and what it will simply pass along as program data.

In any mode the Control-Break keystroke is never data but is a signal that causes the current process to be terminated. In cooked mode, however, the control-C keystroke is treated the same as control-break, while in raw mode it's just a control character. The keystrokes that are signals only in cooked mode include:

- Control-C for break
- Control-S for display pause
- Control-P for printer echo
- Enter (or another "turnaround" character) as end of line.

The *KbdStringIn* call discussed below also operates differently in raw and in cooked modes.

11.4.3 The Turnaround in Cooked Mode

The Keyboard Status data structure contains the current turnaround character. This is the character that marks the end of a logical input line when reading in cooked mode. Appearance of the turnaround character ends input for a *DosRead* or a *KbdStringIn* in cooked mode. The character may be a two-byte symbol to accommodate a DBCS keyboard translation.

11.4.4 Reporting Shift Changes in Raw Mode

The current shift state also appears. This is a truncated version of the shift state returned by *KbdCharIn* and discussed below. If reflects the state of the keyboard LED indicators, and if it is set with *KbdSetStatus*, the indicator lights will change to match.

When the keyboard is in cooked mode, makes and breaks of the shift, Control, and Alt keys have only indirect effects. They change the interpretation of the other keys, and the current shift state is reported along with an input character, but the event of a shift key being pressed is not reported for its own sake.

When the keyboard is in raw mode, shift states are still only indirectly reported. But when the keyboard is put into raw mode, bit 8 of the status mask in Figure 11.1 may also be set on. This requests that any change of shift state be treated as a character in its own right. When this bit is set, any change in shift state—a make or break of any shift, alt, or control key, or press of the Insert, CapsLock, NumLock, or ScrollLock key—will be put into the keyboard input buffer as a null character (a keystroke record with a null character and a scan code of zero).

11.5 Reading a Sequence of Keystrokes

The *KbdStringIn* call takes a buffer and a desired length, and reads a number of keystrokes. It behaves differently in raw and cooked modes.

In cooked mode, the call reads keystrokes until the buffer is filled or until the operation hits the Enter key (or whatever has been set as the turnaround character). Only in cooked mode, the buffer may be filled beforehand with a template string, that is, a previous entry that the operator may rework using the edit keys like F1, F3, Insert and Del. As a result, the standard editing keys can't be returned as data.

Template editing is effectively disabled on the first call to *KbdStringIn* after a *KbdOpen* or *KbdFlush*. After a *KbdStringIn* call has successfully completed, the prepared template will be usable on the next call. In effect, template editing is disabled on a program's first string-input call, since the command shell flushes the keystroke buffer when starting a program.

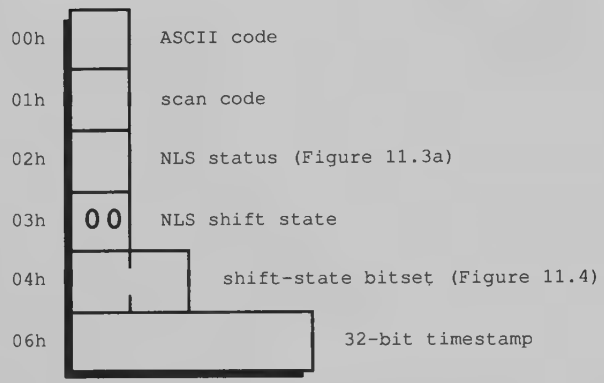
In raw mode, *KbdStringIn* reads whatever keystrokes were present in the buffer. If it is asked to wait for a full count, it reads until it has received exactly as many keystrokes as were requested, no matter how long that takes. The Enter key (or other turnaround character) is just data in raw mode, as are editing keys.

The *KbdStringIn* call does not return shift-change reports even if they have been requested in the keyboard status mask.

11.6 Direct Keyboard Input

In the end, direct keyboard input is not complicated. A call to *KbdCharIn* or *KbdPeek* returns the next input to a logical keyboard in the form depicted in Figure 11.2. This data structure is used in a number of system calls, sometimes alone and sometimes embedded in a larger structure.

Figure 11.2
Keyboard Data Record



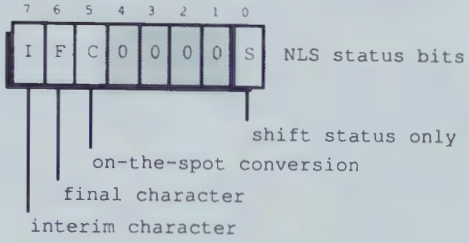
11.6.1 The ASCII Code Byte

The record contains three main items of information. The first is the ASCII code. This is the end product of the keyboard translation of the keystrokes pressed in respect to shift and accent keys. It may be standard ASCII, for example an *X* for an upshift-X key, or a *CAN* code for a control-X key. It might be a value between 128 and 255, for example a code of 148 for ö produced by accent processing, or code 176 produced by holding Alt and keying 1-7-6.

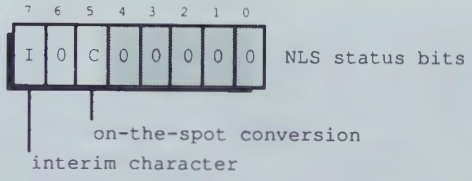
When a function key has been pressed, there is no corresponding ASCII value. The ASCII code byte is set to zero to indicate this, and the program should take this as a signal to examine the scan code byte that follows.

When one of the new function keys, added to the keyboard with the PC/AT, is pressed, there is still no ASCII data value. The use of an "extended" key is signalled with an ASCII code value of 224, 0e0h. The program should take this as a signal to examine the scan code byte. (In the event a frisky user should hold down Alt and key 2-2-4, the ASCII code will be 0e0h and the scan code will be zero.)

Figure 11.3
The NLS Status Bits



(a) As returned in a keyboard data record.



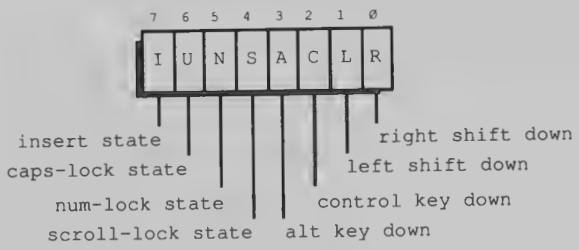
(b) As passed when requesting a change in NLS status.

11.6.2 The Scan Code Byte

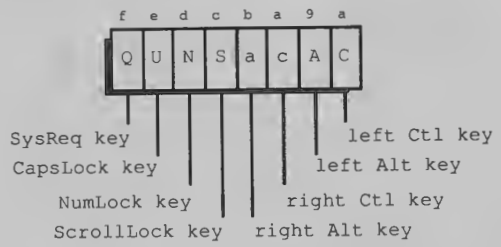
The second item in a keystroke record is the scan code. This is not the original scan code generated by the keyboard hardware, but the end product of the device driver's translation. In many cases it simply reflects the key that was pressed. For instance (see Table 11.1, page 344) the *X* key produces the identical scan code regardless of the shift state. However, function keys, keypad keys, and the top row of digit keys all yield different scan codes depending on the state of the shift keys.

When Function keys are pressed, the ASCII code will be 00h or 0e0h, and the scan code tells the program what key was pressed. Keys with the same label produce the same scan code. For example, the down-arrow key on the keypad and the "extended" down-arrow key both produce scan code 80 (050h). They are distinguished by their ASCII codes, if the program needs to distinguish them. When the keyboard status is set to report shift changes, a change in shift state is returned as a keystroke with zero in both the ASCII code and the scan code bytes.

Figure 11.4
The Keyboard Shift-State Bitset



(a) Least-significant byte of word, lowest address in storage



(b) Most-significant byte of word, higher address in storage

11.6.3 The Shift State

The third item in a keystroke record is the shift state. This is a set of 16 bits that shows the exact shift state at the time the key record was finished (see Figure 11.4). The least significant byte of this field reflects the important shift states: insert, capslock, control, alt, and shift. It even distinguishes between left and right shift keys. This shift information is applicable to any IBM-compatible keyboard.

The more significant byte of the shift bitset reflects shifts available on later, enhanced keyboards, and reflects the make state of some keys. It allows the program to distinguish between the left and right control and alt keys on an

enhanced keyboard. (Allowing a program to actually treat left-alt-X differently from right-alt-X if it wishes.)

This byte also contains the only available input of the SysReq key. That elusive key does not itself generate a keyboard datum. However, the most significant bit of the shift state is set if the SysReq key was *held down* during the entry of the keystroke that did produce the record! Similarly, the down state of the CapsLock, NumLock, and ScrollLock keys is reflected by bits in the key record. This allows a program to use these four keys as *additional* shift keys if it desires—and if its users will tolerate having to "press Alt-CapsLock-X to exit," as the instructions might read.

11.6.4 The Timestamp

The last four bytes of the keystroke data record contain a time value. It's a 32-bit integer, set from the millisecond clock that is available in the Global information segment returned by *DosGetInfoSeg*. The difference between the timestamps of two consecutive key records is approximately the time between keystrokes in milliseconds. This might be useful for game programs.

11.7 Keystroke Translation and Code Pages

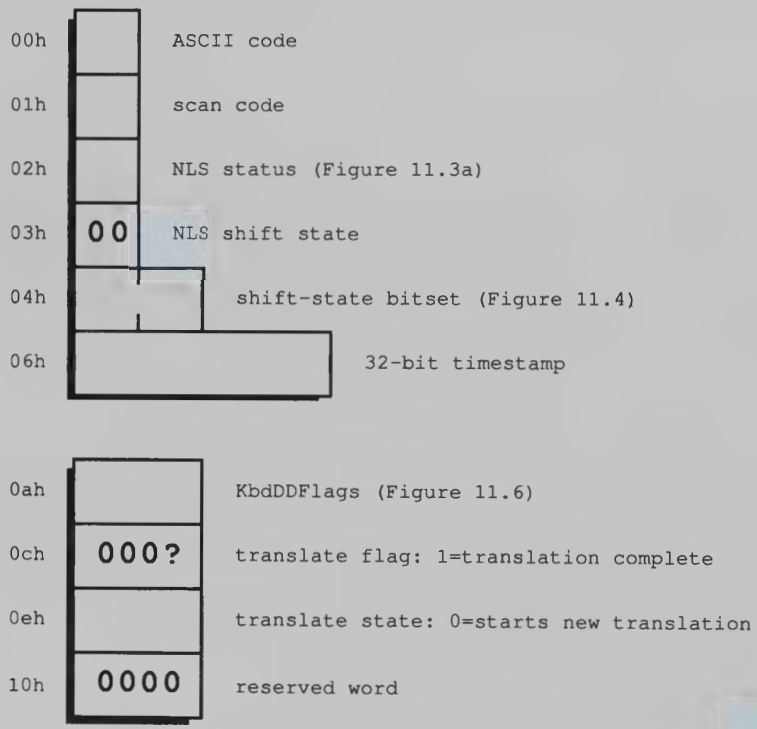
The *KbdGetCP* call returns the code page number now in effect for a logical keyboard. This will often be zero, meaning the default translation built into OS/2. Next most often, it will be the code page number mentioned first in the *codepage=* statement in the system configuration file. Less commonly it may be one of the other numbers listed in that statement. For that to be true, the *KbdSetCP* call must have been used to set a different code page number.

There is yet another way to set the keyboard code page: provide your own. The *KbdCustCP* call takes a code page in storage and installs it in a logical keyboard. The code page format is quite complicated; twenty very dense pages in the *Microsoft OS/2 Device Drivers Guide* are devoted to explaining it (and they are seriously in error). It would take considerable effort and experimentation to create a code page from scratch. It would not be quite as difficult to copy an existing code page and modify it in small ways. There seems little point to this, however, since the *KbdCharIn* call returns almost all information a program could want and, if Echo mode is off, the program is free to interpret the input keystrokes in any way, including retranslation.

11.8 Program-Driven Translation

The translation functions of the keyboard device driver are available through the *KbdXlate* call. This call operates on a data structure that contains a key data record and other fields; see Figure 11.5.

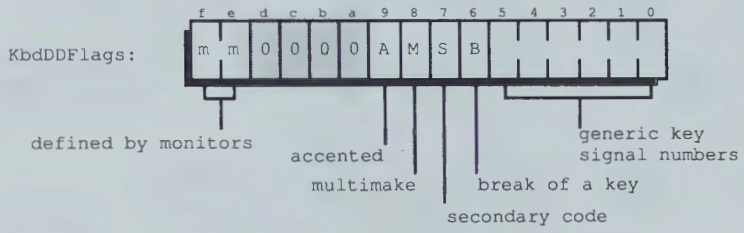
Figure 11.5
Augmented Keyboard Data Record Used in KbdXlate



To begin a translation, set zero in the Translate State field, set the shift state field appropriately, and put a scan code in the scan-code field. This would be a hardware scan code as shown in Table 11.3. (This table has been placed at the end of the chapter owing to its bulk.)

The *KbdXlate* call will update the ASCII code field and the contents of the Keyboard Device Driver Flag Word (see Figure 11.6). field. If the translation is complete, a 1 will be returned in the Translation Flag word; if more scan codes are needed to complete a character (perhaps for an accent) a zero is returned there.

Figure 11.6
Keyboard Device-Driver Flag Word



The Device Driver Flag word shows how the sequence of scan codes has been interpreted by the translation function. In particular its least-significant six bits contain a signal code that shows whether or not a logical signal like Break or Pause or Reboot has been processed. The possible signal codes are shown in Table 11.4.

Table 11.4
Generic Key-Signals Seen in KeyDDFlag Words

Number (dec)	Number (hex)	Meaning
0	00	No special action needed
1	01	Keyboard hardware acknowledgement (AT scan code 0fah)
2	02	Prefix scan code, precedes secondary key (AT scan code 0e0h or 0e1h)
3	03	Keyboard hardware overrun (AT scan code 0ffh)
4	04	Keyboard hardware resend (AT scan code 0feh)
5	05	Reboot key signal (Ctl-Alt-Del)
6	06	Dump key signal (Ctl-Alt-NumLock, twice)
7	07	Shift-change signal; contents other than shift state should be ignored.
8	08	Pause signal (Ctl-NumLock)
9	09	Pseudo-pause (Ctl-S in cooked mode)
10	0a	Wake-up (end pause state)
11	0b	Invalid accent combination seen
12-15	0c-0f	Reserved
16	10	This keystroke is an accent and will be applied to the next keystroke yet to come; do not process it.
17	11	Break signal (Ctl-Break)
18	12	Pseudo-break (Ctl-C in cooked mode)
19	13	Print-screen signal (Shift-PrtSc)
20	14	Printer-echo signal (Ctl-PrtSc)
21	15	Pseudo-printer-echo signal (Ctl-P in cooked mode)
22	16	Print-flush signal (Ctl-Alt-PrtSc)
23-47	17-2f	Reserved, no meaning
48	30	Change in Shift or NLS state commanded by IOCtl, contents other than Shift and NLS to be ignored.
49	31	Written data, produced by DosWrite to file opened as "KBD$"; scan codes are zero.
50-62	32-3e	Reserved, no meaning
63	3f	Scan code with respect to shift state is undefined, or invalid inserted data.

The translation function is of most use to registered replacement keyboard functions, but it can be used in a test program to probe into or check out the keyboard code page.

11.9 Replacing Keyboard Functions

Most of the keyboard system calls can be replaced with your own code. The replacement code takes over the functions of the replaced calls for the entire screen group. The calls that can be replaced are shown in Table 11.5.

Table 11.5
Index Numbers of Replaced Keyboard Functions

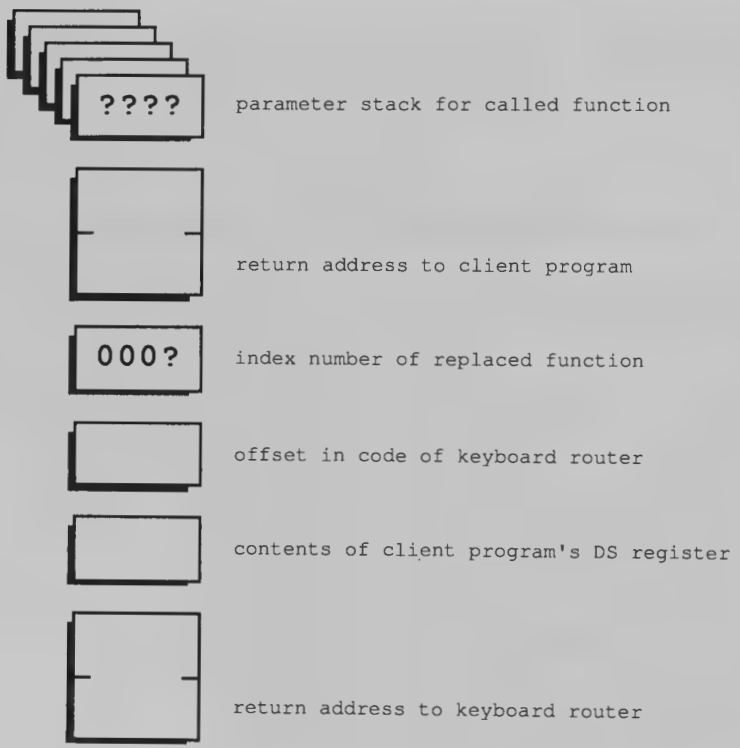
Number	Function
0	KbdCharIn
1	KbdPeek
2	KbdFlushBuffer
3	KbdGetStatus
4	KbdSetStatus
5	KbdStringIn
6	KbdOpen
7	KbdClose
8	KbdGetFocus
9	KbdFreeFocus
10	KbdGetCP
11	KbdSetCP
12	KbdXlate
13	KbdCustCP

The index number of a replaced function is its bit number in the registration bit mask, and a word parameter to the replacing procedure code.

The process is called *registering* a replacement. It is done via *KbdRegister*, and rescinded with *KbdDeRegister*. One or more replacement functions may be registered at a time, but only one registration call is allowed to be in effect at a time within a screen group.

The code for all replaced calls is contained in a single procedure. Its entry point is called whenever any of the replaced functions are called by code anywhere in the screen group. When it is entered, the stack is arranged as shown in Figure 11.7.

Figure 11.7
Stack Layout Entering Replacement Keyboard Function



The client program (the one that called the keyboard function) has stacked parameters for it. Below these on the stack is the client's return address, then some words stacked by the keyboard router, which is the system code that intercepts keyboard function calls and routes them to replacements. One of these words is the index number of the replaced function.

There are a number of restrictions on the code that is registered as a replacement. First, since it may be entered from more than one program, it must be located in a dynamic link library. Only dynalink code segments can appear in the address space of more than one program.

Second, the code must be *reentrant*. That is, it must permit concurrent execution by more than one thread (this follows from its being available to more than one program). If it uses static data, it must serialize access to it with semaphores (not critical sections, as these are only effective within a single process).

One simple way of serialization has already been provided. The *KbdSynch* call is used to request ownership of a semaphore that is unique to the screen group. The semaphore is always cleared when the replacement procedure returns to the router. If the replacement procedure calls *KbdSynch*, the semaphore will act as a key to the code of the procedure. Only one thread will be able to enter it at a time.

Finally, the code cannot use the replaced keyboard functions itself and so must carry out their operations by calling the keyboard device driver with *DevIOCtl*. Because of this, and because of the variable stack layout and the need for reentrancy, the replacement function will probably be written in assembly language.

Table 11.1
Scan Codes Returned by Key and Shift

key	normal		shifted		control		alt	
	dec.	hex.	dec.	hex.	dec.	hex.	dec.	hex.
esc	1	01	(same)		(hot)		(hot)	
1	2	02	(same)		---		120	78
2	3	03	(same)		(same)		121	79
3	4	04	(same)		---		122	7a
4	5	05	(same)		---		123	7b
5	6	06	(same)		---		124	7c
6	7	07	(same)		(same)		125	7d
7	8	08	(same)		---		126	7e
8	9	09	(same)		---		127	7f
9	10	0a	(same)		---		128	80
0	11	0b	(same)		---		129	81
-	12	0c	(same)		(same)		130	82
=	13	0d	(same)		---		131	83
\	43	2b	(same)		(same)		---	
backspace	14	0e	(same)		(same)		---	
tab	15	0f	(same)		(same)		---	
q	16	10	(same)		(same)		(same)	
w	17	11	(same)		(same)		(same)	
e	18	12	(same)		(same)		(same)	
r	19	13	(same)		(same)		(same)	
t	20	14	(same)		(same)		(same)	
y	21	15	(same)		(same)		(same)	
u	22	16	(same)		(same)		(same)	
i	23	17	(same)		(same)		(same)	
o	24	18	(same)		(same)		(same)	
p	25	19	(same)		(same)		(same)	
[	26	1a	(same)		(same)		---	
]	27	1b	(same)		(same)		---	
enter	28	1c	(same)		(same)		---	
a	30	1e	(same)		(same)		(same)	
s	31	1f	(same)		(same)		(same)	
d	32	20	(same)		(same)		(same)	
f	33	21	(same)		(same)		(same)	
g	34	22	(same)		(same)		(same)	
h	35	23	(same)		(same)		(same)	

j	36	24	(same)		(same)		(same)	
k	37	25	(same)		(same)		(same)	
l	38	26	(same)		(same)		(same)	
;	39	27	(same)		---		---	
'	40	28	(same)		---		---	
`	41	29	(same)		---		---	
\	43	2b	(same)		(same)		---	
z	44	2c	(same)		(same)		(same)	
x	45	2d	(same)		(same)		(same)	
c	46	2e	(same)		(same)		(same)	
v	47	2f	(same)		(same)		(same)	
b	48	30	(same)		(same)		(same)	
n	49	31	(same)		(same)		(same)	
m	50	32	(same)		(same)		(same)	
,	51	33	(same)		---		---	
.	52	34	(same)		---		---	
/	53	35	(same)		---		---	
space	57	39	(same)		(same)		(same)	
f1	59	3b	84	54	94	5e	104	68
f2	60	3c	85	55	95	5f	105	69
f3	61	3d	86	56	96	60	106	6a
f4	62	3e	87	57	97	61	107	6b
f5	63	3f	88	58	98	62	108	6c
f6	64	40	89	59	99	63	109	6d
f7	65	41	90	5a	100	64	110	6e
f8	66	42	91	5b	101	65	111	6f
f9	67	43	92	5c	102	66	112	70
f10	68	44	93	5d	103	67	113	71
f11	133	85	135	87	137	89	139	8b
f12	134	86	136	88	138	8a	13	8c
Home	71	47	(same)		119		77	---
up	72	48	(same)		---		---	
PgUp	73	49	(same)		132		84	---
left	75	4b	(same)		115		73	---
pad-5	---		76	4c	---			---
right	77	4d	(same)		116		74	---
End	79	4f	(same)		117		75	---
down	80	50	(same)		---			---

PgDn	81	51	(same)	118	76	---
Insert	82	52	(same)	---		---
Del	83	53	(same)	---		---
pad-plus	78	4e	(same)	---		---
pad-star	55	37	(hot)	---		---
pad-minus	74	4a	(same)	---		---

Table 11.2
Returned Scan Codes in Numerical Sequence

scan code dec.	hex.	key indicated
1	01	esc
2	02	1
3	03	2
4	04	3
5	05	4
6	06	5
7	07	6
8	08	7
9	09	8
10	0a	9
11	0b	0
12	0c	-
13	0d	=
14	0e	backspace
15	0f	tab
16	10	q
17	11	w
18	12	e
19	13	r
20	14	t
21	15	y
22	16	u
23	17	i
24	18	o
25	19	p
26	1a	[
27	1p	]
28	1c	enter
29	1d	(Control)
30	1e	a
31	1f	s
32	20	d
33	21	f
34	22	g
35	23	h
36	24	j
37	25	k

scan code dec.	hex.	key indicated
38	26	l
39	27	;
40	28	'
41	29	`
42	2a	(left shift)
43	2b	\
44	2c	z
45	2d	x
46	2e	c
47	2f	v
48	30	b
49	31	n
50	32	m
51	33	,
52	34	.
53	35	/
54	36	(right shift)
55	37	pad-star
56	38	(Alt)
57	39	space
58	3a	(Caps Lock)
59	3b	f1
60	3c	f2
61	3d	f3
62	3e	f4
63	3f	f5
64	40	f6
65	41	f7
66	42	f8
67	43	f9
68	44	f10
69	45	(Num Lock)
70	46	(Scroll Lock)
71	47	Home
72	48	up
73	49	PgUp
74	4a	pad-minus

75	4b	left
76	4c	sft-pad-5
77	4d	right
78	4e	pad-plus
79	4f	End
80	50	down
81	51	PgDn
82	52	Insert
83	53	Del
84	54	sft-f1
85	55	sft-f2
86	56	sft-f3
87	57	sft-f4
88	58	sft-f5
89	59	sft-f6
90	5a	sft-f7
91	5b	sft-f8
92	5c	sft-f9
93	5d	sft-f10
94	5e	ctl-f1
95	5f	ctl-f2
96	60	ctl-f3
97	61	ctl-f4
98	62	ctl-f5
99	63	ctl-f6
100	64	ctl-f7
101	65	ctl-f8
102	66	ctl-f9
103	67	ctl-f10
104	68	alt-f1
105	69	alt-f2
106	6a	alt-f3
107	6b	alt-f4
108	6c	alt-f5
109	6d	alt-f6
110	6e	alt-f7
111	6f	alt-f8
112	70	alt-f9
113	71	alt-f10
114	72	(n.a.)
115	73	ctl-left
116	74	ctl-right
117	75	ctl-End
118	76	ctl-PgDn
119	77	ctl-Home
120	78	alt-1
121	79	alt-2
122	7a	alt-3
123	7b	alt-4
124	7c	alt-5
125	7d	alt-6
126	7e	alt-7
127	7f	alt-8
128	80	alt-9
129	81	alt-0
130	82	alt-hyphen
131	83	alt-=
132	84	ctl-PgUp
133	85	f11
134	86	f12
135	87	sft-f11
136	88	sft-f12
137	89	ctl-f11
138	8a	ctl-f12
139	8b	alt-f11
140	8c	alt-f12
192	e0	(secondary)
250	fa	(acknowledge)
255	ff	(overrun)

These are the scan codes returned by *KbdCharIn*, *KbdPeek*, or *KbdXlate* when the indicated keystrokes are received. Entries shown in parens are seen only by keystroke monitors; they are never returned on input.

Table 11.3
Input Scan Codes in Numerical Sequence

key pressed	scan dec.	code hex.	key pressed	scan dec.	code hex
esc	1	01	k	37	25
1	2	02	l	38	26
2	3	03	;	39	27
3	4	04	'	40	28
4	5	05	`	41	29
5	6	06	left shift	42	2a
6	7	07	\	43	2b
7	8	08	z	44	2c
8	9	09	x	45	2d
9	10	0a	c	46	2e
0	11	0b	v	47	2f
-	12	0c	b	48	30
=	13	0d	n	49	31
backspace	14	0e	m	50	32
tab	15	0f	,	51	33
q	16	10	.	52	34
w	17	11	/	53	35
e	18	12	right shift	54	36
r	19	13	pad-star	55	37
t	20	14	left alt	56	38
y	21	15	space	57	39
u	22	16	CapsLock	58	3a
i	23	17	f1	59	3b
o	24	18	f2	60	3c
p	25	19	f3	61	3d
[	26	1a	f4	62	3e
]	27	1p	f5	63	3f
enter	28	1c	f6	64	40
left control	29	1d	f7	65	41
a	30	1e	f8	66	42
s	31	1f	f9	67	43
d	32	20	f10	68	44
f	33	21	NumLock	69	45
g	34	22	ScrollLock	70	46
h	35	23	Home	71	47
j	36	24	up	72	48

PgUp	73	49	Del	83	53
pad-minus	74	4a	SysReq	84	54
left	75	4b	(n.a.)	85	55
sft-pad-5	76	4c	(n.a.)	86	56
right	77	4d	f11	87	57
pad-plus	78	4e	f12	88	58
End	79	4f	(secondary)	224	e0
down	80	50	(resend)	254	fe
PgDn	81	51	(overrun)	255	ff
Insert	82	52			

These scan code numbers, when input to KbdXlate, will be interpreted as the keys shown.

12

Mouse Input

There is no need to be intimidated by the mouse. Yes, the little device is unfamiliar to many programmers. Yes, it is often associated with the most complex software systems ever devised, the windowing environments of the OS/2 Presentation Manager and others. But the device itself is not complicated, and the OS/2 system calls to use it are no more complicated than those for keyboard use.

If the system has a mouse, it is available in every screen group. In a screen group controlled by the Presentation Manager, the mouse is operated by the Presentation Manager on behalf of all programs. In other screen groups, any program may open the mouse and use it.

A program will ordinarily use the mouse in three phases. First it will open the mouse and inquire into its characteristics. Then it will prepare the mouse for use. In the main part of the program, it will read mouse events one at a time as they occur and interpret them.

Reference: screen group 1.2.1

12.1 How the Mouse Works

The mouse is a hand-held, motion-sensing device with a couple of buttons. The operator slides the mouse on a surface, and the mouse hardware detects the sliding motion. Some mice do this mechanically, others optically.

12.1.1 Mouse Interrupts and Events

Whenever the mouse detects motion, or when the operator presses or releases one of its buttons, the mouse signals the computer with an interrupt.

The interrupt is handled by a mouse device driver. The device driver reads the mouse device's input to find out what has befallen it. It pushes the information onto a first-in-first-out queue of events. There is a separate queue of mouse events for each screen group. The device driver puts the mouse event into the queue of the screen group that happens to be in the foreground at the time.

The device driver also keeps track of the position of a mouse pointer on the display screen. The pointer may not be visible, but even so, whenever the mouse reports motion, the device driver updates the pointer's coordinates. There's a separate pointer for each screen group. Whenever the mouse moves, device driver updates the coordinates of the pointer for the screen group that's visible.

12.1.2 Mouse Units

The smallest motion that the mouse can detect is called a *Mickey*. The relationship between a Mickey and a standard unit of length varies with the kind of mouse; some are more sensitive than others. One of the OS/2 system calls returns the number of Mickeys per centimeter of motion for the current mouse. The Microsoft Bus Mouse I'm using reports 8 Mickeys per centimeter (a bit more than 20 per inch).

Whenever the mouse detects a Mickey of movement, it interrupts the computer. Some mice are connected to the system bus and cause the interrupt directly. Others are connected to a serial port; these signal by writing a byte to the serial port and the serial port causes the interrupt.

Since the Mickey is quite small, there can be a lot of interrupts and therefore a lot of reported events. However, when the mouse is moved smoothly and fast, either the mouse or the device driver batches the moves and reports many Mickeys in one event.

It doesn't much matter whether the batching is done in the device driver, the mouse hardware, or elsewhere. There are many kinds of mice with a variety of connection methods, and it is much safer to regard the mouse hardware and device driver as a single black box that behaves in certain predictable ways.

12.1.3 The Mouse Pointer

The mouse device driver keeps the coordinates of a pointer for each screen group. The coordinate units depend on the display mode. In text mode, the pointer is located by character row and column; in graphics modes it is located by pixel row and column. In any mode, the location is kept within the bounds of the screen dimensions.

12.1.3.1 The Pointer Drawing Routine

Before the mouse has been opened the mouse pointer is only a logical position, not a visible mark on the display. After a program opens the mouse, it may ask that the pointer be made visible. From then on, the mouse device driver will call a drawing routine to remove and redraw the pointer image whenever the pointer location changes.

The pointer drawing routine must act as a subroutine of the mouse device driver. For technical reasons, it must itself be loaded as a device driver (with a *device=* statement in the configuration file). The default pointer drawing code is named *POINTDD.SYS*, and it loads as a device named *POINTER$*. There is provision for using other drawing routines, but none are supplied with the basic system.

Reference: device drivers ch. 13

12.1.3.2 The Pointer in Text Mode

When the display screen is in a text mode, the mouse pointer is displayed as a single character cell, usually with background and foreground colors inverted. As the operator moves the mouse, the device driver calls the drawing routine to erase and redraw the pointer image in the display buffer.

The pointer image is created by modifying the attribute byte of a character cell in the display buffer. Unfortunately, the code that scrolls the display does not know about the mouse pointer. If the screen scrolls under the pointer, the character cell with inverted colors will scroll right along with it and will be moved to another row. A normal character will scroll into the pointer's location, leaving it dark.

The pointer drawing routine won't know about the scrolling, of course. The next time it is called to erase and redraw the pointer, it will only see that the cell at the

old pointer location has changed. It will leave that alone, and reverse the colors of the cell at the new pointer location. The result is that scrolling creates a trail of residual pointer cells across the display. To avoid this, disable drawing of the mouse pointer before the screen is allowed to scroll. It is not difficult to disable the mouse pointer, scroll, and enable the pointer again afterward.

References: display modes 10.1.2; display attribute byte 10.4.2.2; scrolling 10.4.2.5

12.1.3.3 The Pointer in Graphics Mode

The pointer drawing routine supplied with version 1.0 of the system can only cope with text modes. When the display is in a graphics mode, it gives up and the pointer remains undrawn. The possible solutions to this are discussed below.

References: display modes 10.1.2

12.1.3.4 Excluding the Pointer

The program may specify a rectangular patch, from one character or pixel up to the size of the screen, within which the mouse pointer may not be drawn. The pointer's location is updated as usual, but when it falls within the excluded area, the pointer is not drawn. The pointer disappears as it reaches the edge of the patch, and reappears on the far side. This creates an interesting visual illusion. When the mouse is moved rapidly, the pointer seems to slide under or behind the reserved patch, and the patch takes on an appearance of solidity and dimension.

If only one rectangular region of the screen is being scrolled, excluding the pointer from that region would prevent scrolling the mouse pointer.

12.2 Opening the Mouse

The mouse becomes available to any process that opens it. The real, physical mouse hardware is controlled by its device driver. The device driver creates a virtual mouse—the appearance of a mouse, represented by a pointer location and an event queue—for each screen group. Any process may open a handle to the virtual mouse for its screen group. This is done with a call to the *MouOpen* service.

The *MouOpen* call gives access to the mouse. It takes the name of a pointer drawing routine, and returns a handle that represents the mouse in other operations. *MouOpen* also initializes the mouse to a neutral state.

12.2.1 Is a Mouse Available?

However, there may not be a mouse installed in the system. A mouse is not required to run OS/2, and many systems do not have one. If there is no hardware mouse, or if the mouse device driver was not loaded, *MouOpen* will return an error code.

The mouse is part of a screen group, but some processes are not associated with a screen group. Programs started with a Detach command, or with a *run=* command in the configuration file, or with *DosExecPgm* option 4 (detached execution) have no screen group. The *MouOpen* service will return an error code to such a program.

Finally, in the Presentation Manager's screen group, the *MouOpen* call will have been preempted by the Presentation Manager, and will almost certainly return an error code. The Presentation Manager controls the mouse on behalf of all the programs running in its windows.

In short, a program should expect that *MouOpen* may return a nonzero error code, and if it does, the program should be prepared to interact with the operator in some other way.

Reference: *DosExecPgm* 6.1.2

12.2.2 Choosing a Drawing Routine

MouOpen takes one input parameter: the name of a pointer drawing routine. Ordinarily this parameter will be given as a null far pointer (a doubleword of zeros), indicating that the program will use the default system pointer. Alternatively it may be the address of a character string *POINTER$*, which is the name of the default pointer drawing routine loaded as *POINTDD.SYS*.

At least in version 1.0, this drawing routine can't handle graphics display modes. That won't be a problem for a program that uses text-mode displays. A program that puts the display into a graphics mode will have to find another way to display the mouse pointer.

The first solution would be to procure or prepare a substitute drawing routine, one that can handle pixel graphics. Such a routine must be loaded as a device driver at configuration time; then the device name that it defines can be named to *MouOpen*. The pointer drawing routine operates as a direct subroutine of the mouse device driver. It is called by the device driver at interrupt time. Accordingly, it, like other device drivers, must be able to execute in either Protect or Real mode and will probably be written in assembly language. In short, creation of a pointer drawing routine is not a job for the faint of heart.

Another solution is to do the pointer drawing in the program itself. The program will be reading and interpreting mouse events in any case, so updating one small patch of the screen when the mouse moves will not be a large burden. While it takes a certain amount of time for a mouse move to be reported in the event queue and received by the program, the delay should be no more than barely perceptible to the operator.

A third solution is to write a mouse device monitor. This is discussed in the chapter on monitors.

References: display modes 10.1.2; device drivers ch. 13; device monitors ch. 16

12.2.3 The Initial State

When the mouse is opened, it is initialized. Its pointer location is set to the middle of the screen (row 12 column 40 of the common 80-by-25 display format). The entire screen is excluded, so that the pointer won't be drawn and the program will have to take action to make the pointer visible. The pointer is initialized to a default shape (inverted colors, in text mode). The event mask, which determines what events will be entered to the queue, is set to allow all possible events. The queue itself is emptied.

12.3 Preparing the Mouse

After opening the mouse, the program will probably want to inquire about the mouse's features, and will probably need to prepare it for use by setting the device status, pointer shape, and scale factor.

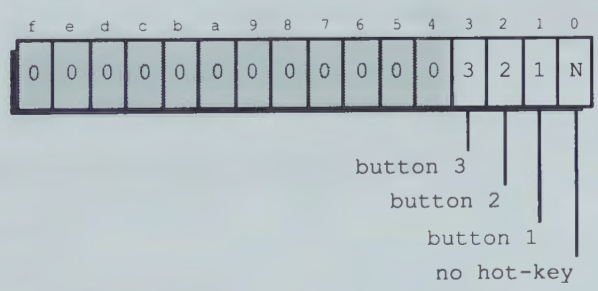
12.3.1 Finding Out about the Mouse

There are several system calls that return information about the mouse. The *MouGetNumButtons* call returns the number of buttons (one, two or three) on the mouse. The same information can be inferred from the mask returned by *MouGetEventMask*, discussed later.

The sensitivity of the mouse itself is returned by *MouGetNumMickeys*. This call returns the number of Mickeys that will be counted in a movement of one centimeter. This information can be some help in setting the scale factor, discussed just below.

It is possible that some combination of buttons has been reserved by the Session Manager as a hot key signal. The *MouGetHotKey* call returns a bit map shown in Figure 12.1. If bit zero of that word is set, there is no reserved hot-key combination and all mouse buttons are available for the application to use. If bit zero is zero, the combination of other bits shows what button-press has been reserved by the Session Manager.

Figure 12.1
Session Manager Hot-Key Definition from MouGetHotKey



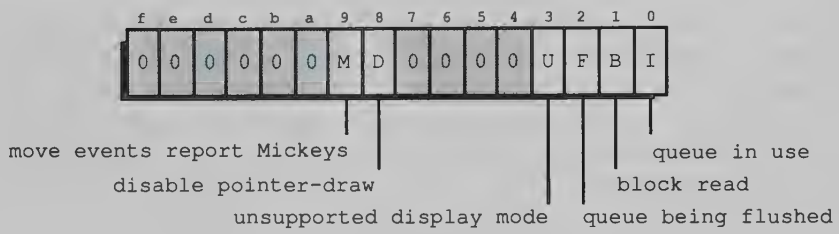
12.3.2 Getting and Setting Device Status

The *MouGetDevStatus* call returns a word containing a few status bits, as shown in Figure 12.2. Bits zero, 1 and 2 of this word reflect transient events of no great interest to an application.

Bit 3, however, is significant. It is set whenever the pointer drawing routine cannot cope with the current display mode. In version 1.0 of OS/2, this bit is set following *MouOpen* whenever the display is in a graphics mode.

As is often the case in OS/2, the mouse status is maintained with a complementary pair of calls. The *MouSetDevStatus* call takes the word returned by *MouGetDevStatus*. The preferred way to change the status is to get it, change the bits of interest, and reset it.

Figure 12.2
Mouse Device Status Bits



In Figure 12.2 there are only two bits that are settable. The program may set bit 8 to disable pointer drawing, either permanently or temporarily. This is the bit to set prior to scrolling the display. Reset it afterward to reenable pointer drawing.

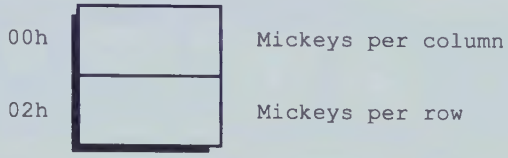
Bit 9 is used to choose between two kinds of event reporting. It will be discussed later.

12.3.3 Getting and Setting the Motion Scale

The device driver is notified whenever the mouse moves as little as a Mickey, either horizontally or vertically. But a Mickey is a small distance. How many Mickeys ought the mouse to shift before the mouse pointer on the screen moves?

The answer depends on the display mode and on the preferences of the operator. The mouse scale factor specifies the ratio between Mickeys and pointer units. The *MouGetScaleFact* call returns the scale information in the structure shown in Figure 12.3.

Figure 12.3
Data Structure Used by MouGet/SetScaleFact



These two counts show the number of Mickeys that must be counted before the mouse pointer will be moved one unit horizontally or vertically. In any text mode with the Microsoft Bus Mouse, *MouOpen* initializes the scale to 16 Mickeys per row and 8 per column. That is, the mouse must report 16 Mickeys of vertical motion, 2 cm., before the device driver will move the mouse pointer one row vertically.

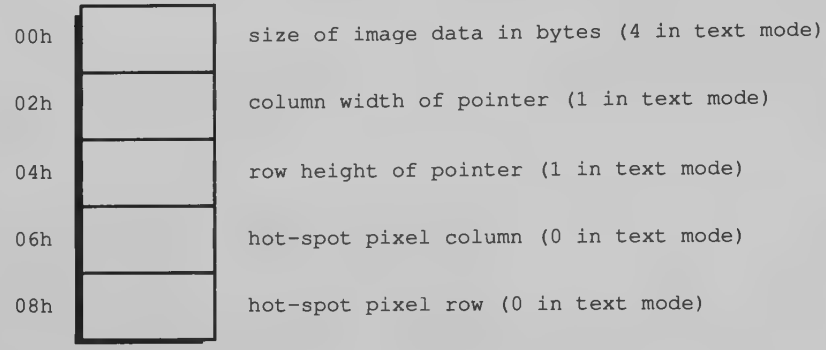
The *MouSetScaleFact* call is the complement; it takes the structure shown in Figure 12.3 and sets a new scale factor.

Larger numbers in the scale make the mouse less sensitive; smaller ones make it more touchy. People vary widely in their ability to coordinate their hands and eyes; I have watched some users who had great difficulty in moving a mouse pointer to a spot on the screen, while others seem never to miss. I am a notably uncoordinated person and I find the default setting touchy; it is much easier to tag a specified character on the screen when the default values are doubled. It is a good idea to provide some way for the user to adjust the scale factor to suit personal needs.

12.3.4 Getting and Setting the Pointer Shape

The mouse pointer image is specified by a series of image bytes plus the information kept in the structure shown in Figure 12.4. The pointer image data can be retrieved using *MouGetPtrShape*, regardless of whether the pointer is being drawn or not.

Figure 12.4
Data Structure Used by MouGet/SetPtrShape



The *MouGetPtrShape* call returns the image bytes in a buffer and fills in the structure shown. The *MouSetPtrShape* call is its complement; it takes the same parameters, possibly updated, and installs them.

To understand the pointer data you must understand one key point: the pointer's image does not *replace* the existing screen data; it is always drawn by *modifying* the existing screen data at the pointer location. The image bytes specify how the modification is to be done.

The mouse pointer is drawn by *AND*ing some image bytes against the existing screen image, and then *XOR*ing other image bytes against the result of the *AND* operation. The combination of these two operations allow any of four possible outcomes, as shown in Table 12.1.

Any bit of the screen image that falls under the pointer may be set to zero, or to one, or preserved or inverted. The image bytes returned by *MouGetPtrShape* comprise an *AND* mask the size of the pointer image, followed by an *XOR* mask of the same size.

12.3.4.1 The Text-Mode Pointer Image

When the display is in a text mode, the pointer image is always a single character cell. A cell occupies two bytes in the display buffer, the first for the character and the second for its display attribute. Therefore in text mode there are always four image bytes: a two-byte *AND* mask followed by a two-byte *XOR* mask.

Table 12.1
Effect of AND and XOR Masks on Image Bit

Bit of AND Mask	Bit of XOR Mask: zero	Bit of XOR Mask: one
zero	zero	one
one	preserved	inverted

The *MouOpen* call initializes the text pointer image to the four bytes:

```
ffh ffh 00h 77h
```

The *AND* mask contains all 1-bits. The first byte of the XOR mask has all 0-bits, so (referring to Table 12.1) the bits of the character byte in the cell will be preserved.

The second *XOR* byte inverts the three bits of each of the color choices, so that if the colors were zero and seven (white on black), they will afterward be seven and zero. That might be expected to create reverse-video black on white letters, but it usually creates a solid white block. I have had better results changing the second *XOR* byte to 7fh, so as to invert the brightness of the foreground as well as its color. This almost always leaves the character under the pointer visible regardless of the original colors.

Reference: display attributes 10.4.2.2

12.3.4.2 The Graphics-Mode Pointer Image

When the display is in a graphics mode (and when the pointer drawing code supports it) the size of the pointer is specified in pixels in the second and third fields

of Figure 12.4. The form of the *AND* and *XOR* masks is not well defined in this case. The bits in each mask correspond to the pixels of the image, probably in row by row order. It is not clear whether each row should be filled out to a byte boundary. This seems likely but I have been unable to test it; so, take care.

When the graphics mode requires multiple bit-planes, as when one of the enhanced modes of the EGA or VGA is in use, there is supposed to be a pair of masks for each bit plane, in sequence.

In graphics modes the image will usually be larger than the coordinate unit (the pixel). In that case, where should the pointer image be drawn relative to the coordinate location of the pointer? The last two fields of Figure 12.4 answer this question. The location of a hot spot is specified by relative pixel rows and columns within the pointer image. That pixel will be drawn on the pointer location and the rest of the image will fall around it.

Reference: display modes 10.1.2

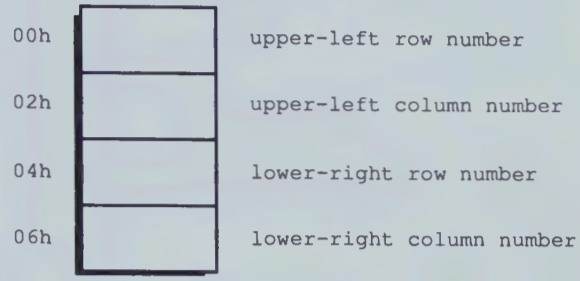
12.3.5 Revealing and Excluding the Pointer

The last step of preparing the mouse is to reveal the mouse pointer. *MouOpen* leaves the pointer excluded from the whole screen. A call to *MouDrawPtr* cancels any excluded area; afterward—provided drawing is possible in the current display mode and permitted by the device status bits—the mouse pointer will be drawn on the screen and will track the motions of the mouse.

The program may also want to exclude the pointer from some part of the display. This is done with *MouRemovePtr*, which doesn't remove the pointer but does define a region in which it can't be drawn. This call takes the data structure shown in Figure 12.5.

This data structure was incorrectly documented in some early OS/2 documentation. The layout shown above has been tested: each pair of fields gives the display coordinates of one corner of a rectangular area. The corners must fall in the bounds of the current display mode.

Figure 12.5
Exclusion-Area Data for MouRemovePtr



12.4 Using the Mouse

Once the mouse has been prepared, the program has nothing to do but wait for an event to occur. The device driver puts each event into a queue and the program takes them out. A program that prepares the mouse, waits on events, and updates a full-screen display to reflect those events, is included in the Disk Appendix to this book.

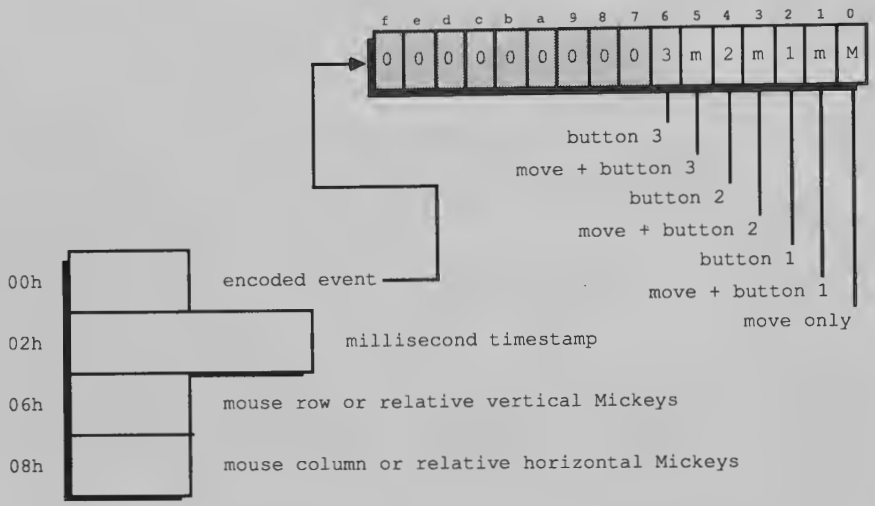
12.4.1 Reading an Event

The *MouReadEventQue* call returns the next event from the queue, presenting the data in the structure shown in Figure 12.6.

The bit mask in the first word explains what happened. These bits may appear in any combination, including the combination of all-zero bits. That would not in-

dicate a non-event; it would mean that a button that had previously been pressed had now been released.

Figure 12.6
Event Data Structure from MouReadEventQue



12.4.1.1 Choosing Events

It may be that only certain events are important to the application: only moves with a button held down, for example. The *MouGetEventMask* call returns a word in the same format as the event mask shown in Figure 12.6, but with a 1-bit for every reportable event. *MouOpen* sets this mask to every event that is possible for the attached mouse. If the mouse has only one or two buttons, event bits for the missing buttons will be set to zero.

The mask returned by *MouGetEventMask* may be altered and updated using *MouSetEventMask*. Regardless of what events are reported in the event queue, the device driver continues to track the pointer location and redraw the pointer when that is permitted.

12.4.1.2 The Event Timestamp

The 32-bit timestamp in the second field is a copy of the system's millisecond timer at the time the event was queued. This count of milliseconds is actually updated on every timer tick, or once every 32 milliseconds.

Applications that use the timestamp will most often be interested in the interval between two mouse events. This can be encoded by subtracting the first timestamp from the second and dividing the result by 32 (shifting right five bits). No precision is lost, and the result can usually be stored in a 16-bit unsigned word. However, if the operator goes to lunch between events, the interval will exceed the capacity of a word. If longer intervals are meaningful, all should be stored in 32-bit form; otherwise, ones that overflow 16 bits can be converted to 65,535 and called infinity.

12.4.1.3 The Event Position

The last two fields in Figure 12.6 contain position information. Normally these will contain the mouse pointer coordinates at the time of the event (at the end of the move, if a move is reported).

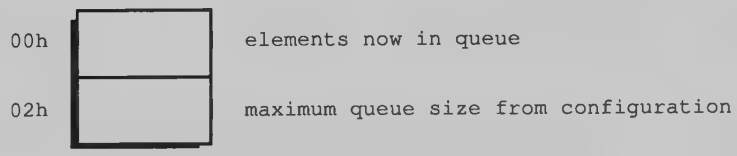
If bit 9 of the device status byte is set (using *MouSetDevStatus*), the event data will reflect relative Mickeys of motion. In that case these fields contain signed integers, where negative numbers mean up or left and positive ones mean down or right.

This mode of reporting greatly increases the number of move events that are reported. It would be used when the program draws its own mouse pointer.

12.4.2 Checking Queue Status

The program can find out how many events are queued by calling *MouGetNumQueEl*; it returns two words as shown in Figure 12.7. The first word shows how many events are in the queue waiting to be read. The second word shows how many the queue will hold; this number is set in the configuration file.

Figure 12.7
Data Structure Returned by MouGetQueStatus



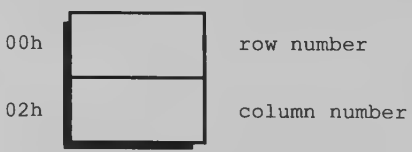
In a lightly loaded system the program will have no difficulty keeping up with the mouse. However, the system load and therefore a program's share of the CPU cannot be predicted. A sophisticated application might want to check the queue and, if it is more than three-fourths full, raise the priority of the thread that processes mouse events.

The mouse queue can be emptied entirely with a call to *MouFlushQue*. This is done by *MouOpen*, so that the queue starts out empty. A program might do it again when it changes modes of operation or changes the screen format.

12.4.3 Changing the Pointer Location

If mouse events are reported with their absolute coordinates, the program will always know where the mouse pointer is. When events are reported with relative Mickeys, the pointer location can still be obtained from a *MouGetPtrPos* call. This reports the current pointer coordinates in the structure shown in Figure 12.8.

Figure 12.8
Data Structure Used in MouGet/SetPtrPos



The location reported by *MouGetPtrPos* may be quite different from the location at which the last-read event occurred. Even if the *MouGetPtrPos* call is coded directly after the *MouReadEventQue* call, the program can be suspended for an unknown amount of time between the two calls—an interval during which the mouse may be moving! In a lightly-loaded system the interval will usually be

insignificant, but a program cannot *presume* it will always run in a lightly-loaded system. If the coordinates of an event are important, they must be reported in the event record.

The program can specify a new pointer location with *MouSetPtrPos*. It takes the same structure as *MouGetPtrPos* returns.

12.5 Closing the Mouse

When the program is finished with the mouse, it can release the handle for other uses with a *MouClose* call. A *MouClose* is done automatically when the process ends, if the mouse is still open then.

12.6 Replacing Mouse Functions

Most of the mouse system calls can be replaced with your own code. The replacement code takes over the functions of the replaced calls for the entire screen group. The calls that can be replaced are shown in Table 12.2. Ordinarily this mechanism would be used by a program that creates and manages the execution environment for other programs, a shell like the OS/2 Presentation Manager.

The process is called *registering* a replacement. It is done via *MouRegister* and rescinded with *MouDeregister*. One or more replacement functions may be registered at a time, but only one registration call is allowed to be in effect at a time within a screen group.

The code for all replaced calls is contained in a single procedure. Its entry point is called whenever any of the replaced functions are called by code anywhere in the screen group. When it is entered, the stack is arranged as shown in Figure 12.9.

The client program, the one that called the mouse function, has stacked parameters for it. Below these on the stack is the client's return address, then two words stacked by the mouse router, the system code that intercepts mouse function calls and routes them to replacements. One of these words is the index number of the replaced function as given in Table 12.2. This can be used to select the sub-procedure that handles the specific function.

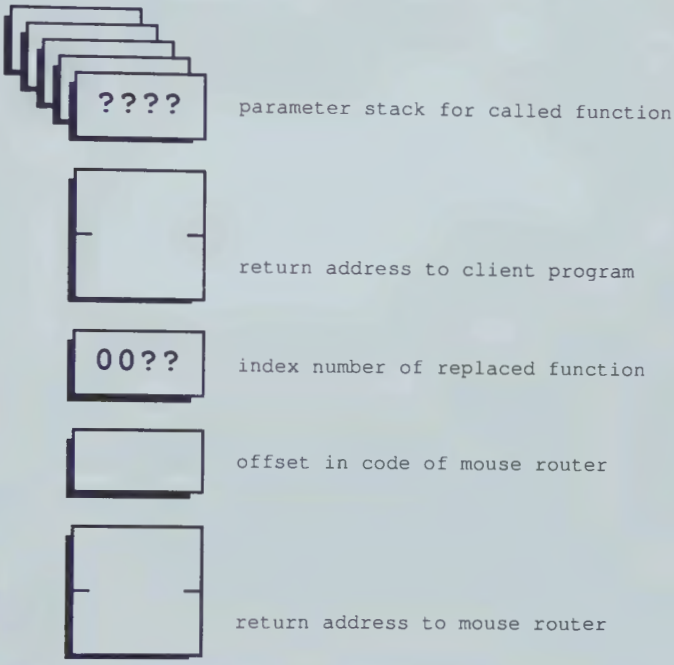
Usually, a replacement mouse function will be invoked when a process in the screen group makes that call. This will be true whether the screen group is visible or in the background. On the other hand, processes in other screen groups are

Table 12.2
Index Numbers of Replaced Mouse Functions

Number	Function
0	MouGetNumButtons
1	MouGetNumMickeys
2	MouGetDevStatus
3	MouGetNumQueEl
4	MouReadEventQue
5	MouGetScaleFact
6	MouGetEventMask
7	MouSetScaleFact
8	MouSetEventMask
9	MouGetHotKey
10	MouSetHotKey
11	MouOpen
12	MouClose
13	MouGetPtrShape
14	MouSetPtrShape
15	MouDrawPtr
16	MouRemovePtr
17	MouGetPtrPos
18	MouSetPtrPos
19	MouInitReal
20	MouFlushQue
21	MouSetDevStatus

The index number of a replaced function is its bit number in the registration bit mask, and a word parameter to the replacing procedure.

Figure 12.9
Stack Layout Entering Replacement Mouse Function



unaffected; they go on using the standard mouse calls or, possibly, the replacements that are registered in their screen groups.

There are a number of restrictions on the code that is registered as a replacement. First, since it may be entered from more than one program, it must be located in a dynamic link library. Only dynalink code segments can appear in the address space of more than one program.

Second, the code must be *reentrant,* that is, it must permit concurrent execution by more than one thread (this follows from its being available to more than one program). If it uses a segment of static data it must serialize access to it with semaphores (not critical sections, as these are only effective within a single process). The *MouSynch* call is available for this purpose; it performs a *DosSemRequest* function on a system semaphore that is unique to the screen group. The mouse router releases the semaphore when the replacement procedure returns.

Finally, the code (obviously) cannot use the replaced mouse functions itself and so must carry out their operations by calling the mouse device driver with *DevIOCtl*, and/or by performing direct hardware operations from an I/O privileged segment. Because of this, and because of the variable stack layout and the need for reentrancy, the replacement function will probably be written in assembly language.

13

Device Drivers

In OS/2 the fundamental operations of input and output are performed by *device drivers*. Ordinarily a program uses their services indirectly, by reading and writing files. However, it is possible to communicate directly with device drivers, and there are some hardware operations that can only be accomplished that way. In this chapter we discuss device drivers in general; in later ones we will see how to call on device drivers to control the serial ports, and to read and write disk volumes in physical units.

13.1 Device Drivers

A device driver is a subroutine of the OS/2 kernel. Its purpose is to exercise physical control over one hardware device or a class of similar devices. The driver is compiled and linked separately from OS/2; it may even be supplied by a different manufacturer.

A device driver embodies the knowledge of how a device is operated. Since the driver is separate from the OS/2 kernel, the kernel itself has no dependencies on the characteristics of any particular brand of hardware. Hardware makers can supply device drivers to integrate their devices smoothly into OS/2. For example, the IBM release of OS/2 contains two sets of device drivers, one for the PC/AT series of machines and one for the PS/2 series. It also contains five different mouse device drivers, one for each of five common makes of mouse.

13.1.1 Driver Installation

A driver is installed during system configuration. A set of base device drivers are loaded automatically; these include the keyboard, screen and printer drivers.

Others are not loaded automatically but as a result of being named in a *device=* statement in the system configuration file. The drivers for mouse and serial port must be specified this way.

Here is the approximate sequence of events that take place during system initialization. The system starts up with enough built-in hardware support to read the disks and write to the screen. It reads the *CONFIG.SYS* file and processes all the statements *except* the *device=* statements; those it saves, in their given order, for last.

Finally the *device=* statements are processed in order, beginning with imaginary ones that name the printer, keyboard and screen drivers. Each driver is stored in an executable file which contains a compiled, linked, program. The program is loaded in much the same way an application is loaded. However, device drivers are loaded into low storage and thus take up space in the Real-mode area (if one is enabled).

A device driver program is subject to many constraints, and its interface to the operating system is quite different from that of a normal application program. For instance, it must consist of just one code segment and one data segment, and the first bytes of the data segment must contain a control block, called a device driver header, that describes the driver to the system. Device drivers may only use a few of the OS/2 system calls that are used by applications, but they have access to a set of device driver helper services that are not available to application programs.

After a driver has been loaded, the system invokes it to initialize itself. During this time the driver will reset and test the hardware it operates, and it may set up to trap interrupts from that device. Thereafter, the system will call on the driver whenever a file is opened or closed to its device, and whenever the system needs an operation done on the device that the driver controls.

Reference: loading executable files ch. 4

13.1.2 Block and Character Drivers

There are two types of device drivers. Drivers that control *block-oriented* devices like disks are quite different from ones that control *character* devices like the keyboard or serial port. The block devices are designated by drive letters (actually drive numbers), and each driver is expected to support one volume of the OS/2 file system. OS/2 treats each block device as an expanse of random-access stor-

age on which it can store directories and files. OS/2 calls on a block-oriented device driver to read and write sectors of data, but it retains control of the content of the sectors.

Drivers that control *character* devices do not have letters but names like *COM1*. There is one device and one device driver per name (several similar devices may share a single code segment but there is in effect a device driver per device). OS/2 expects a character device to act as a source or sink for data, but does not expect it to handle data in any fixed quantities and does not expect the data to have any particular content.

13.2 Device Driver Operations

A device driver is expected to provide services to OS/2. The system invokes the driver with a request to read or write some number of bytes of data, or to query the status of the device, or to perform some control function like seeking to a particular track. The driver performs the operation and reports the result.

The system also invokes the driver to inform it whenever a file is opened or closed to its device. Some device drivers make no use of this information; others may keep a count of open files; others may do special initialization when a file is opened or closed.

Often a device operation will take a lot of time. Seeking another track on a disk, for example, might take several hundred milliseconds, and the CPU could do work for many threads in that time. The interface between the system and device driver allows for this. The driver may start an operation and return. Later, when the end of the hardware operation is signalled by an interrupt, the driver can signal OS/2 that the work is complete.

OS/2 uses device driver services to implement its file system. When a program calls *DosOpen* to open a disk file, OS/2 will call on the disk device driver to read the sectors of one or more directories. When the program calls *DosRead* to get data from the file, OS/2 will call on the device driver to read the sectors that contain the data for that part of the file. What the program sees as a few simple file operations may be converted into dozens of calls on the disk device driver.

A program may also open a file to a particular character device. During configuration, OS/2 notes the names of all character device drivers. If the *DosOpen* service is given a suitable file pathname (no drive, or directory path, or filetype) and a suitable flag (open if existing, fail if new), it checks the filename against the

device names for a match. If there is a matching name, the file is identified with that character device. *DosRead* and *DosWrite* calls for that file will be converted directly into read-data and write-data requests to that device driver.

References: *DosOpen* 8.7; *DosRead* and *DosWrite* 8.11

13.3 Fake Device Drivers

Device drivers have a special, privileged position; they are trusted subroutines of the operating system and they may trap hardware interrupts. Some optional parts of OS/2 have been coded as device drivers when they do not, strictly speaking, drive a device. In particular, routines that trap the DOS and BIOS calls issued by real-mode programs are written as device drivers. During installation they set up to trap the real-mode INT instructions that some MS-DOS applications use.

Three such fake device drivers are distributed with OS/2 version 1.0: *ANSI.SYS*, *EGA.SYS*, and *POINTDD.SYS*. The first, *ANSI.SYS*, intercepts screen output calls in real mode and implements the ANSI graphics functions. The same support is built into the *VioWrtTTY* call used by protect-mode programs; *ANSI.SYS* serves real-mode programs only.

The driver in *EGA.SYS* monitors the graphics functions performed by Real-mode programs. It tries to perform the display-save and -restore functions for the compatibility screen group, that in protect mode would be done by a watchdog thread using the *VioSaveRedrawWait* function.

The *POINTDD.SYS* driver intercepts Real-mode requests to move the mouse pointer, and saves and restores the mouse pointer for the Real-mode screen group.

Although the fake device drivers have the form of character drivers and therefore have names (*EGA$* for instance) they don't respond to read and write requests. Attempts to open files to their names will fail.

References: ANSI support 10.3.4; display save and redraw 10.5.4; mouse pointer drawing 12.1.3

13.4 The IOCTL Mechanism

The file interface based on *DosOpen*, *DosRead* and *DosWrite* is limited to input and output; it does not allow for any control functions. That is not usually a

problem; most applications only need to read and write while leaving the controlling to OS/2. But sometimes a program needs to control a specific device in ways beyond simple input and output. How can it do this? One way would be to write the code to operate the device into the program. But that would require a code segment with I/O privilege, and would embed dependencies on specific hardware values in the program. Besides, there are some things (like trapping interrupts and locking segments into storage) that only device drivers, not programs, may do.

Must a program become a device driver in order to control a device? No. There is a way in which a program may send requests for service directly to a the device driver. If the device driver will cooperate, the program may achieve control of the device without being dependent on the specific hardware. This method of requesting I/O control services is abbreviated IOCTL. People with nimble tongues try to pronounce it as a word; the rest pronounce it "I/O control."

13.5 Generic I/O Control

A program sends a request to a device driver by issuing the *DosDevIOCTL* call. The call asks the system to pass information to a device driver and return its return code. A set of device driver control functions has been standardized with OS/2. Device drivers are expected to support the generic IOCTL functions that are appropriate to their devices. Some of these are useful to application programs and will be discussed in later chapters.

13.5.1 Selecting the Device Driver

One parameter of *DosDevIOCTL* is a file handle as returned by *DosOpen*. A device driver will be associated with any open file. A disk file will be associated with the disk device driver, while a file opened to a character device will be associated with the driver for that device. Thus a file handle implicitly specifies a device driver; the rest of the parameters of *DosDevIOCTL* will be passed to that device driver. The intent of this design is that *DosOpen*, *DosRead* and *DosWrite* will be used as the main data path to the file, while *DosDevIOCTL* using the same handle may be used for special control functions to set the mode of the file.

Reference: DosOpen 8.7

13.5.2 Category and Function

Two of the parameters of *DosDevIOCTL* are a *category* code and a *function* code. Together these specify what service is wanted. A summary of the generic categories appears in Table 13.1.

Table 13.1
Summary of Generic IOCTL Categories

Category Code (hex)	Type of functions
01	Serial port operations
02	(not defined)
03	Video display operations
04	Keyboard operations
05	Printer operations
06	(not defined)
07	Mouse operations
08	Disk control of logical volumes
09	Disk control of physical drives
0a	Register a monitor
0b	Monitor operations

The functions under category 1, Serial Port Operations, are discussed in Chapter 14. Some of the functions under categories 3, 4, 5, and 7 are restricted to the Session Manager. Those that are not, are duplicates of functions that are available as system calls. These categories aren't documented further in this book.

The functions under categories 8 and 9 have to do with physical I/O to disk volumes. A few of these functions are discussed in Chapter 15. The last two categories are related to device monitors, but are duplicated by system functions. Device monitors are discussed in Chapter 16.

13.5.3 Parameter and Data "Packets"

The remaining two parameters of *DosDevIOCTL* are two far pointers. Ordinarily these will address a *parameter packet* and a *data packet*. The form and content of the packets is different for every category and function. Generally, the parameter packet will contain control or descriptive information, and the data packet will contain the data that is the subject of the service.

Sometimes the parameter packet is unimportant because the service is completely described by the category and function codes. Sometimes the data packet is unimportant because no data is exchanged. Different drivers have different ways of handling unneeded packets. The serial port driver insists that an unused packet be represented by a null address (a doubleword of zero). On the other hand, the disk driver wants an unused data packet passed as a valid far pointer to a doubleword of zero in storage.

The disk appendix to this book contains a program that will issue a *DosDevIOCTL* call based on command-line parameters, and report the results.

14

Serial Port Operations

The design of the IBM PC/AT allows for two *serial ports* (the IBM PS/2 Model 50 allows three). A serial port is a general, flexible hardware interface, and a great variety of devices may be attached to one. Unfortunately the use and programming of the serial ports is extremely complicated. In order to use the serial ports from a program you must first understand the port and the specific device attached to it very well. Then the OS/2 device driver will probably allow your program to do what it needs to do.

14.1 Use of the Serial Ports

The serial ports are named *COM1* and *COM2* (and *COM3* in the PS/2 Model 50). These are in fact the names of their device drivers, but the names also imply the hardware addresses of the devices themselves and the interrupt levels they employ. In conversation and in the documentation of software, the names are often used to refer to the hardware.

In Real mode there are several ways of controlling the serial ports. Programs may access them indirectly by redirection of the standard input and output files, or may open them as files. Programs may call on the ROM BIOS for device control functions. And many programs manipulate the hardware directly. In MS-DOS applications these methods are considered roughly equivalent, so if a program is said to *use COM1* that does not necessarily mean that it opens a file to the name *COM1*. It may very well manipulate the hardware adapter with base address 3f8h directly.

In Protect mode similar options are available. If the serial port device driver (*COM01.SYS* or *COM02.SYS*) has been installed, files may be opened to the

names of the serial ports, and standard files may be redirected to them. The ROM BIOS is not available, but programs may control the ports by IOCTL calls to the device driver, which will be discussed later. And programs may contain segments of I/O privileged code from which they manipulate the serial port hardware directly, though this is definitely not recommended.

Since there are so many ways of getting at the serial ports, it is almost impossible for the system to restrict access to them. If Real-mode programs are permitted in the system at all (*protectonly=no* in the configuration file), there is no way to prevent one of them from manipulating the serial port hardware directly. If I/O privilege is allowed (*iopl=yes* in the configuration file) there is no way of preventing a Protect-mode program from doing so. It's possible to have several programs all trying to control the serial port hardware at once.

There is a system utility called *SETCOM40* that is supposed to restrict the use of a serial port to Real mode only, or Protect mode only. Unfortunately it doesn't have much effect; it certainly doesn't stop programs that operate directly on the hardware in either Real or Protect modes. There is always the possibility of having contention between programs for the use of the serial ports, and it must be left up to the operator to avoid it.

14.2 The RS-232 Interface

The serial ports use a hardware interface that was standardized by the American National Standards Institute (ANSI) a long time ago as computer history runs, under the name RS-232. The original purpose of the interface was to connect an item of data communication equipment—that is, a modem—to an item of data terminal equipment—that is, a terminal—with the aim of communicating data over a telephone line.

At the time the standard was devised, these two kinds of equipment were regarded as very different; they weren't each a handful of chips as now. As a result the RS-232 interface is *asymmetrical*; it is not a connection between equals, but rather an interface between two different kinds of devices, each with a distinct role to play.

The interface has since come to be used between many sorts of devices, and it isn't always clear which one is supposed to play the role of terminal and which the modem. In many cases, there is no modem and no telephone line involved at all, and the features of the interface that cater to telecommunications are not needed. Unfortunately people didn't leave the unneeded features unused; they put

them to use, often in ways contradictory to the original intent of the standard. Every part of the RS-232 interface has to be understood before it can be used.

14.2.1 The Serial Connector

The standard connector for an RS-232 serial port is a 25-pin plug called a DB-25 connector. The serial ports for most OS/2 systems will be brought out to a DB-25 connector, and the pin numbers cited later apply to this connector.

The interface actually uses fewer than half the 25 wires of a DB-25 connector. Some systems, especially portable ones, may try to save space by bringing the serial port lines to a smaller connector, perhaps a DB-9. Of course, that means that before you can connect such a port to any standard device, you will have to fabricate or buy a custom cable with a DB-25 at one end and the smaller connector at the other.

Whenever a custom cable is made, it is all too likely that—owing to poor documentation, misunderstanding of the RS-232 interface, or plain ignorance—the pins of the two connectors will not be wired together properly. Then an RS-232 breakout box will be needed to debug the interface. A breakout box cables between two RS-232 devices and displays the values of the interface signals on small lights so that the confusion may be made visible. It also permits pins to be patched to each other in different combinations, so that confusion may be compounded.

14.2.2 The DCE and the DTE

One of the devices connected by an RS-232 interface must play the role of the Data Terminal Equipment, or DTE. In the case of an OS/2 system, that will always be the computer. The serial ports on a PC/AT or PS/2 Model 50 are wired for the DTE role and are terminated in male sockets, as the DTE actor's port should be.

The other member of an RS-232 partnership must play the role of Data Communications Equipment, or DCE. A DCE device should be equipped with a female DB-25 connector. An external modem, for instance, will always be wired as a DCE (it is literally a communications device, after all) and will have a female connector that can be cabled directly to the male serial port connector on the back of the computer. Or you may have some other kind of device that is wired to play the DCE role; it should plug into the serial port without trouble.

Sometimes one of the COM devices is not actually a serial port with a connector on the outside of the computer, but a modem board plugged into the computer's bus. That's fine; the device ought to respond as if it were an external modem connected to the serial port by a cable. It's just that the modem, cable and connectors have been telescoped into the system unit.

Problems arise when the computer needs to be connected to another device that is also wired to be a terminal, a DTE; for example, a plotter, a printer, a digitizing tablet, or another OS/2 system's serial port. The other DTE won't connect directly to the serial port. We'll examine the ways around this after reviewing the signal lines of the interface.

Table 14.1
The Eight Signal Lines of the RS-232 Interface

Pin No.	Signal	Purpose
2	data	Outbound data from DTE (terminal) to DCE (modem)
3	data	Incoming data from DCE (modem) to DTE (terminal)
20	DTR	DTE says it is ready (Data Terminal Ready)
6	DSR	DCE says it is ready (Data Set Ready)
4	RTS	DTE asks permission to transmit (Request To Send)
5	CTS	DCE gives permission to transmit (Clear To Send)
8	DCD	DCE says it has a connection (Data Carrier Detect, also called RLSD, Receive Line Signal Detect)
22	RI	DCE says it has an incoming call (Ring Indicator)

Worse problems can arise from sloppily-designed products. I have seen a printer from a well-known maker of computers and instruments that for no obvious reason was wired as a DCE—and had a male connector. It took more than a few minutes to connect that machine! There was a voice synthesizer (happily now off the market) that had an RS-232 interface that wouldn't work as either DCE or

DTE. It appeared to have been designed in the hope that it would connect with any type of serial port, but without a custom cable or a well-patched breakout box it worked with nothing.

14.2.3 The RS-232 Signals

In almost all applications there are only eight significant wires in the RS-232 interface. They are summarized with their standard pin numbers in Table 14.1 .

14.2.3.1 The Data Lines

A DTE device like the computer will always transmit on pin 2 and receive in pin 3, while a DCE device will always listen on 2 and transmit on 3. When a DCE is cabled to a DTE, this works fine. But if two DTEs are plugged together, or two DCEs are plugged together, no data can be exchanged. They will both transmit on the same wire, and both will listen to the other wire on which will be no data.

The simplest RS-232 connection uses only pins 2 and 3, and no other signals. It's often possible to transfer data between two devices using only these two wires.

14.2.3.2 The DTR Signal

Each of the actors has a signal line to say that it is alive and ready for operations. The DTE device—that is, the computer—asserts a signal on pin 20, DTR, when it is ready for business.

This signal will often affect the operation of the connected device. A true DCE device connected to the serial port will be designed to do nothing so long as DTR is false. After all, if the terminal is inoperable, perhaps not powered on, it can't receive on pin 3 while any data that appears on pin 2 is probably garbage.

Other devices that might be connected to the serial port may ignore the state of DTR. There are still others that use inverted logic, taking a false condition of DTR as meaning ready.

14.2.3.3 The DSR Signal

The DCE device—the modem or whatever is connected to the serial port—should assert a signal on pin 8, DSR, when it is ready for operations. The name of the signal, Data Set Ready, originated when "data set" was a term for a modem.

When the connected device really is a DCE, the state of DSR is important. If the DCE isn't operable, there's no point in sending data to it and any incoming data is probably spurious. However, if the connected device isn't really a DCE, the state of the DSR line may be unimportant or erroneous.

14.2.3.4 The RTS and CTS Signals

The DTE (computer) requests permission to transmit by asserting the RTS signal, and the connected device gives permission by asserting CTS. These signals are left over from the antique days of half-duplex modems that could only send or receive, but could not do both simultaneously. It took a significant amount of time to turn the line around from receiving to sending. The keyboard operator would push a request or attention button; the terminal would raise RTS; a fraction of a second later the modem would respond on CTS; then the terminal would unlock its mechanical keyboard so that the operator could begin typing.

Modern modems can usually transmit and receive simultaneously, so there's often no need for the DTE to ask permission so long as DSR is true. But if the attached device isn't a modem, the CTS signal might be irrelevant or it might reflect some other condition (like a printer's being out of paper). Or it might simply be the RTS signal, wrapped back on the CTS wire, so that a request on RTS creates the CTS signal.

14.2.3.5 The DCD (RLSD) Signal

When the connected device is really communications equipment, it may be important for the computer to know whether there is a connection or not. If there's no connection, there may be no point in transmitting and no valid data to receive.

When the connection is through a telephone line, it is complete when the modem detects the acoustic tone emitted by another modem. The tone is called the data carrier, and so long as the modem can hear it, it will assert a true signal on the Data Carrier Detect line. The connection might be through a digital medium; then DCD would be based on some other indication that the hook-up was com-

plete. Or the connected device might not be a modem at all. In that case, the DCD line might be irrelevant or it might represent some other condition at the whim of the designer.

Data Carrier Detect, DCD, is an old name for this signal. In later revisions of the standard the same signal has been renamed Receive Line Signal Detect, RLSD.

DCD is less important now when most modems are smart, that is, have their own CPU and program. Even when DCD is false, there is still a need to transmit local instructions to the modem itself, and the modem's responses are valid data. In fact, the only way to make DCD true may be to transmit a dialing command to the modem. If DCD false were to prevent output, DCD could never be made true!

14.2.3.6 The RI Signal

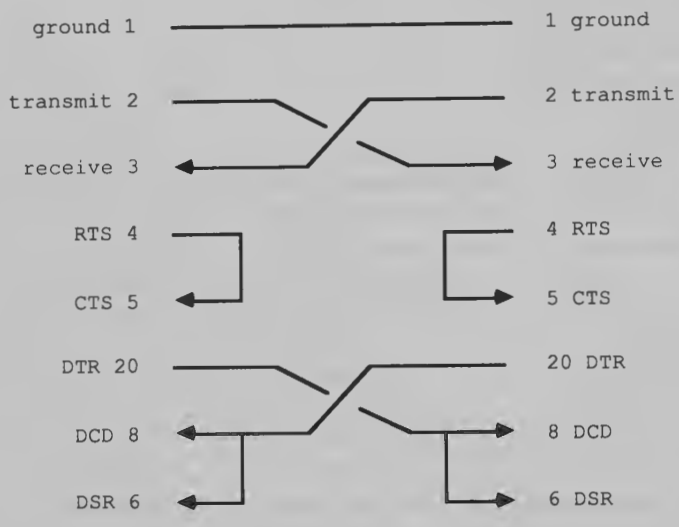
When the connected device is a modem set up to answer the phone—as it might be if the computer is acting as a bulletin board system—the Ring Indicator signal indicates that the modem has detected the telephone bell signal of an incoming call; someone has dialed the computer and the phone is ringing.

When the connected device is not a modem, the RI signal is probably unimportant, but it might be hooked up to indicate some other kind of start-up or reset signal.

14.2.3.7 The Null Modem

The RS-232 interface is not symmetrical, but it can be forced into a symmetrical shape by a specially-wired cable called a null modem. A null modem is a way of patching the eight signal lines of one DTE to those of another DTE so that each will treat the other as a modem. The wiring diagram is shown in Figure 14.1.

Figure 14.1
Wiring Diagram for a Null Modem



Many possible wirings can be devised. This arrangement works between two devices that are wired as DTEs, for example two computers.

The data signal lines 2 and 3 are simply crossed, one system's pin 2 going to the other's pin 3, and vice versa.

There are two ways to handle pins 4 and 5, RTS and CTS. They can be shorted on each side, so that each DTE's RTS becomes its own CTS. This permits each system to transmit at any time; that's usually not a problem. Alternatively, the pins 4 and 5 can be crossed so that RTS from one device becomes CTS at the other device. This works when each DTE asserts RTS true all the time as a matter of course; the signals cross and become constant CTS signals on the other side.

Finally, each system's DTR signal becomes the other's DSR and DCD; as soon as one system is operable the other sees a working, connected modem.

14.2.4 The Data Format

Data is exchanged over an RS-232 connection in units properly called words. These are not the 16-bit words used in computer storage; word here means "the bits representing one character," and to avoid confusion I shall call them characters. The sending device writes the bits of each character, one bit at a time, onto its transmitting line. The receiving device takes them off and reassembles them into a character.

Each character that is sent is framed by stop bits and may be followed by a parity bit. The form of a character and the speed with which it is sent comprise the *line protocol* being used.

14.2.4.1 Signal Levels

On the data lines, a negative voltage is the idle condition; it is called the *mark* or *marking* state and, within a character, means a 1-bit. A positive voltage is the active state; it is called *space* or *spacing* and within a character signals a zero-bit. The names *mark* and *space* come from telegraphy, and are a fossilized remnant of Victorian technology in modern computers.

An idle RS-232 data line will be in the marking state. Ordinarily the line will never stay in the spacing state for more than one character time. A long spell of solid spacing is called a *break* (or reverse break). The serial port hardware will recognize a break as a special event, not a character.

14.2.4.2 Number of Bits Per Character

The number of bits to be used in one character is a matter of agreement between the connected devices. Some devices (telex machines for example) exchange characters of only five bits (they manage to encode an alphabet of fifty-odd characters in five bits by using the shift-in and shift-out characters to shift the printing element from lower to upper case). Others use seven bits (the ASCII standard is a 7-bit code). Eight bits is normal between computers, since most computers use 8-bit bytes in storage. It is possible to transmit 8-bit byte values between computers using 7-bit characters, but the software at each end has to perform some kind of conversion on the data. The Kermit protocol, for example, was designed to transmit 8-bit data over 7-bit lines.

The OS/2 serial port hardware can be set up to transmit and receive any character size from five to eight bits. Only one thing is required: both systems must be using the same number of bits. If they don't, they won't be able to make sense of each other's transmissions.

When less than eight bits are sent in a character, the serial port hardware assigns them to the least-significant bits of a byte, so there is always one byte read or written per character, regardless of the character size.

14.2.4.3 Number of Stop Bits

The receiving device needs some way of detecting the boundary between characters; otherwise it wouldn't be able to parse the stream of bits into characters. The stop bit is a gap in the transmission of bits that sets one character off from the next. The length of the gap may be one bit-time, or one and a half, or two bit times. Most devices work with only one stop bit, but the only effect of having too many is to slow down the rate of transmission.

14.2.4.4 Parity

A parity bit is an error-check; it allows the receiver to detect many transmission errors. The parity bit is an extra bit generated by applying an algorithm to the bits of the transmitted character. The hardware counts the 1-bits in the character. Then, to generate *odd* parity, it appends a parity bit of either zero or one, whichever will make the total of 1-bits an odd number. *Even* parity is the opposite: a parity bit value is sent that will make the count of 1-bits even.

Some equipment requires a parity bit in every character but either doesn't check it (so its value doesn't matter) or else requires it to have a certain fixed value. *Mark* parity is a parity bit that is always set to zero (marking); *space* parity is always set to 1 (spacing). The serial port hardware can generate any of these four kinds of parity, or no parity bit at all.

It is usually essential that both devices agree on the type of parity to be used. However, a device that expects to receive eight-bit characters with no parity will often work successfully with a partner that is sending seven-bit characters with parity, especially mark parity.

Parity is a fairly crude means of error checking. When blocks of data are being sent from one computer to another, the software normally embeds longer, more

elaborate error-detection codes within each data block. There are a variety of methods in common use (the oldest is Ward Christiansen's *XMODEM* protocol; *YMODEM*, *CompuServe-B* and Kermit are others). When embedded error correction is used, the serial port parity bit is usually ignored or not used.

14.2.4.5 Transmission Rate

It is crucial that both devices agree on the rate at which the bits are to be sent. There are standard rates for sending bits. The slowest that the serial port can handle is 110 bits per second; the next fastest is 150 bits per second. These slow speeds are used by old mechanical devices. The remaining speeds are successive doubles of 150 bits per second: 300, 600, 1,200, 2,400, and so on up to 9,600 and even 19,200 bits per second.

These are speeds in bits per second; in fact a speed specifies the length of time that will be given to one bit of data. The time taken for one character's bits depends on the format of the character. An eight-bit character with parity and one stop bit occupies ten bit-times.

The rate of character transmission is important to performance. The serial port hardware will generate a hardware interrupt each time a character is received. From that instant, the CPU has at most one more character time to respond to the interrupt and read the incoming byte value from the hardware. If it doesn't take the character before the next character's bits have arrived, the first character will be lost. This is called a *hardware overrun.*

Suppose that eight-bit characters with no parity and one stop bit are being received at 9600 bits per second. There are nine bit times per character, giving a transmission rate of about 1067 characters per second. The CPU has less than one millisecond to respond to each interrupt. This is probably possible, at least in some systems, but the CPU will not be doing very much work other than responding to interrupts (a CPU with an 8 Megahertz clock rate will average fewer than 2,000 instructions per millisecond). In particular the system won't be spending much time executing the program that is processing data from the serial port, so while the device driver may be able to capture the incoming data without overrun, the receiving program may not be able to process it. The result is a *software overrun.*

14.2.5 Flow Control

Whenever two devices are connected the problem of *flow control* arises: how will the flow of data be controlled so that overrun will not occur? Each device needs some way to tell the other to pause in its transmission. An amazing number of solutions have been used, and the serial port hardware and device driver supports most of them.

There are two sides to the question. The receiver must ask, "how can the sender be made to slow down or stop sending?" The sender must ask, "when is it safe to send?"

14.2.5.1 Input Flow Control

The OS/2 system is always a DTE. When a DTE is receiving, it has only two means of telling the sender to stop, one based on hardware and the other on software.

The hardware solution is to set a false condition on the DTR line. Some printers set DTR false when they are out of paper, for example, and the computer program may be playing the role of a printer to some other system. However, the DTR line only goes as far as the connected device; it doesn't extend across a communications link. It might stop a modem from passing on the characters coming down a telephone line, but it won't stop the remote system from sending those characters.

The software solution is to send a signal that means "please stop" up the line to the remote device. The most common signal is the ASCII control-S character, the one with the value 19 decimal, 13 hex, called *XOFF*. The receiving device, when it needs a pause to catch up, transmits *XOFF*; the remote system should respond by pausing in its transmission.

When the receiving system has caught up, it sends control-Q, value 17 decimal, 11 hex, named *XON*. The remote system resumes sending.

The same logic may be based on other character values. I have seen devices that sent *ENQ* (value 5, control-E) to mean pause and then *ACK* (value 6, control-F) to mean resume.

Considerable time can pass between the moment when the *XOFF* is sent and moment when it is received at the sending system. Telephone links that run

through Earth satellites can incorporate delays of half a second, so as much as one second's worth of characters might arrive before the leading edge of the pause reaches the receiver. That's another thousand characters at 9,600 bits per second (but such a high speed isn't common over telephone links).

14.2.5.2 Output Flow Control

The OS/2 program must always ask, is it safe to transmit? As a DTE device, it has several different ways of pacing its output so as not to overrun the receiver.

Three hardware signals can be used to throttle the DTE's output. A half-duplex modem will prevent the DTE from transmitting by setting CTS false. Some connected devices use the DCD line for the same purpose. Others may use the DSR line. There must be agreement between the sending program and the connected device as to whether these three signals, singly or in combination, will regulate the computer's output.

The simplest method of output flow control is to pace the rate of transmission. The rate of character transmission can be adjusted independently of the bit rate, for instance by using two stop bits instead of one. Or the program might simply write one character at a time with a delay between each. Alternatively characters can be sent in blocks of convenient size, with a delay between blocks. Or a delay can be inserted between logical units like lines (it's common for a time-sharing system to require a delay of a half second after every return character).

Of course, if the remote system expects to be able to request a pause by sending *XOFF* (or some other character), the system must honor its request. However, the program may have issued a *DosWrite* request to send a block of characters. If the *XOFF* is to be honored immediately, it will have to be handled by the device driver in the midst of output. The serial device driver can be asked to do this.

14.3 The Serial Port Device Driver

The serial port device driver is delivered in an executable file named *COM01.SYS* (*COM02.SYS* for PS/2 Models). Unlike other device drivers included in the distributed system, it is not loaded automatically. The serial port device driver—let's call it the serial driver for short—will only be loaded if it is named in a *device=* statement in the configuration file.

Table 14.2
Serial Port Operating Parameters

Parameter	Initial Value	Mode Cmd Can Set?	Changes during DosOpen	Changes during DosClose	IOCTL used to Query	IOCTL used to Set
bit rate	1200	√			61h	41h
bits per character	7	√			62h	42h
parity	even	√			62h	42h
stop bits	1	√			62h	42h
DTR signal value	false		√	√	66h	46h
RTS signal value	false		√	√	66h	46h
transmitting break	off			√	64h	4bh,45h
value of XOFF	13h		√		73h	53h
value of XON	11h		√		73h	53h
substitute for errors	off/00h		√		73h	53h
substitute for break	off/00h		√		73h	53h
strip input nulls	ff		√		73h	53h
write timeout	on/60s	√	√		73h	53h
read timeout	on/60s		√		73h	53h
Input sensitive to DSR	on	√			73h	53h
Input flow control to use						
DTR signal	off	√			73h	53h
RTS signal	off	√			73h	53h
XOFF transmission	off	√[1]			73h	53h
Output flow control to use						
DSR signal	on	√	√		73h	53h
CTS signal	on	√	√		73h	53h
DCD signal	off	√[1]			73h	53h
XOFF receipt	off	√			73h	53h

[1] Always set "off" by the Mode command

The serial driver treats each serial port (*COM1*, *COM2*, and *COM3* if it exists) as a separate device. In effect there is a separate, independent driver for each port. Here I will discuss the serial driver as if there were only one port and one driver,

but everything that is said here applies independently and asynchronously to each serial port.

14.3.1 Operating Parameters

The serial driver maintains status information about many operating parameters of the port. These parameters are set to initial values when the driver is first loaded. All of them can be altered using *DosDevIOCTL* calls. Some of them can be altered by the use of the OS/2 Mode command.

The operating parameters are summarized in Table 14.2. The table displays a confusing mix of line protocol items, hardware signals, and software parameters. Unfortunately, serial port programming actually involves a confusing mix of just those things.

Many of these operating parameters are recorded in a data structure called the Device Control Block, or DCB. One DCB structure is maintained by the device driver for each serial port. There are IOCTLs to retrieve a copy of the DCB and to update the DCB with new contents. These IOCTLs, numbers 43h and 73h, are discussed below. A glance at the rightmost column of table 14.2 will show how many operating parameters the DCB controls.

14.3.2 Open and Close

After initializing, the serial driver waits to hear that a file has been opened to the port. Whenever *DosOpen* opens a file to the name of a serial port, OS/2 calls the device driver to notify it of the open. When *DosClose* is used to close the file, the driver is again called and notified of the close.

The serial driver does not care which processes open files to the port, nor does it care how many files are concurrently open. It does a maintain a count of how many files are open. The count is initially zero; it increments on open and decrements on close.

On what it terms a *first open*, that is, an open that bumps the count from zero to one, the driver initializes some of the operating parameters. On what it terms a *last close*, that is, a close that brings the count from one back to zero, the driver performs some clean-up operations. Opens and closes between these two are only counted; they have no effect on the operation of the serial port.

It may seem strange that the serial driver will accept any number of opens. However, there are good reasons for this policy. First, there may be good uses for it. I can imagine an application in which one process took input from the port while a different process wrote output to it. There is also the chance that a process using the port might start a child process that opened the port, used it, and closed it before returning to its parent.

Second, if exclusive access is needed, *DosOpen* can supply it. The share options specify whether the calling process is willing to share the file with readers, with writers, or with no other processes. If all processes that open a serial port demand exclusive access with the *deny all* share option, only one process at a time will succeed in opening it.

Later I will suggest using the Mode command as a child process to set serial port parameters. The Mode command in version 1.0 apparently opens the serial port with access mode of read. If another process has the serial port open with a share option of *deny all* or *deny readers,* the Mode command will be unable to open the serial port and will report failure. If another process has the file open with a share option *deny none* or *deny write* the Mode command will be able to do its work.

Finally, truly exclusive access is impossible as long as Real mode programs are permitted in the system. There is no way of preventing a Real mode program from operating on the serial port hardware. For example, the BASIC interpreter (Microsoft's GWBASIC or IBM's BASICA) will happily execute an *OPEN COM1...* statement at any time, regardless of *setcom40* and regardless of an share options that a Protect-mode program may use.

References: *DosOpen* 8.7; access modes and share options 8.7.3.1, 8.7.3.2

14.3.2.1 Actions on First Open

On a first open of the port, the serial driver clears the serial port receive hardware of any residual input character. Then it sets several operating parameters.

It sets the state of the DTR and RTS signals according to the policies noted for them in the Device Control Block. Ordinarily these policies will both be *enable on open* (Mode command operands *DTR=on* and *RTS=on*) and the driver will set both signals to true. This simulates the operation of a real terminal when it is

powered up; most terminal devices set DTR true when they are turned on and hold RTS equal to DTR at all times.

One or both of these policies might be *disable on open* (Mode command *DTR=off* or *RTS=off*); then the driver will set the signal or signals to false. This would usually be the case when the program intended to control the state of the signal line(s) directly, using IOCTL 46h.

The driver sets the byte value of *XOFF* to 13h with control-S, and that of *XON* to 11h with control-Q. Applications that use different byte values as flow control characters can change them with IOCTLs as noted in Table 14.2.

The driver disables the features of substituting for input errors and breaks, and clears the substitute values to zero. It also disables the feature of stripping input nulls. These features can be changed with IOCTLs.

The driver clears any flow control state information that might remain from previous operations. In particular it sets its transmission state to show that it has not received an *XOFF*.

The read and write timeout intervals are set to 60 seconds. The read timeout policy is set to the normal *wait for timeout or full count*. The write timeout policy is not changed at open, since the Mode command might have been used to set it (Mode *TO=on* or *off*). The effect of these intervals is discussed below.

14.3.2.2 Actions on Last Close

When the serial driver is notified of a close and its count of open files drops to zero, it cleans up the state of the port. Note that this final close might not have been issued by the same process that issued the first open; many overlapping opens and closes might have been done. However, this close marks the end of all use of the serial port by active processes, at least for the present.

The driver's first action is to discard all pending output data and all unread input bytes. A program that writes to the serial port must take pains to ensure that all of its output has been written before closing the file. There are IOCTL calls that enable it to tell when all data has been written.

If a break is being transmitted, the driver turns it off. If input flow control using *XOFF* was in use, and if an *XOFF* had been sent, an *XON* is sent to unblock the remote system.

Finally, if the DTR and RTS signals are not already false, the driver waits for ten character times (at the current transmission rate) and sets both signals false. This simulates the normal operation of a real terminal being turned off.

14.3.3 Reading and Writing

Once the serial port has had a first open, reading and writing go on independently and asynchronously. The input and output sides of the serial port may be thought of as different, almost unrelated, devices.

Whenever the bits of an input character have been assembled, the serial port hardware causes an interrupt. As quickly as possible OS/2 gives control to the serial device driver, which takes the new character from the hardware and stores it in an *input queue*, a queue of characters received but not yet read by a process. If the driver does not respond quickly enough to the interrupt, a hardware overrun has occurred. If the driver retrieves the character in time, but finds that the input queue is full, a software overrun has occurred. In either case the incoming character is lost.

The serial driver also maintains a *transmit queue*, a queue of bytes waiting to be transmitted. Whenever the hardware finishes sending all the bits of a character, it causes an interrupt. OS/2 dispatches the device driver, and it takes the next character, if there is one, from the transmit queue and sets it into the hardware for transmission. As long as there is data in the transmit queue, the serial driver will keep the output side of the serial port running at full speed.

14.3.3.1 Writing

Output to a serial port is the same as output to any file: a process calls *DosWrite* (or *DosWriteAsync*) with a handle, a buffer address, and a length. OS/2 translates this into a call on the serial driver with a request to write.

The driver queues write requests in the order it receives them. Then, one request at a time, it copies the data from the original buffer into the transmit queue, which is in storage that is private to the device driver. When the transmit queue empties, it copies more bytes until all the bytes of the request have been moved. When the last of these bytes has been written into the hardware for transmission, the driver notifies OS/2 that the write request is finished. OS/2 then allows the

DosWrite (or *DosWriteAsync*) call to complete; the count of written bytes will match the initial length.

When a *DosWrite* call ends, the process's buffer space may be reused since the output data have been transmitted. However, serial output takes time; the output has not actually finished until one character time later.

References: *DosWrite* 8.11.4; *DosWriteAsync* 8.11.5

14.3.3.1.1 Write Timeouts

Transmission can be blocked by several kinds of output flow control. When transmission is blocked, the current write request (and any that arrived after it) are delayed. Ordinarily, output is blocked for only a short time. However, if the attached device fails, the delay could be substantial. What happens then depends on the write timeout policy.

If the write timeout policy is for timeouts (Mode command *TO=on*) the driver will wait for the expiration of the write timeout interval (a number in the DCB, initially 60 seconds), and then give up on the current write request. The error code returned by *DosWrite* will be zero, but the count of bytes written will reflect the number of bytes actually transmitted, and might be zero. The driver starts a new timeout interval as soon as even one byte has been transmitted. The driver also starts a new timeout interval when it begins work on the next write request. That is, if multiple write requests are queued when a timeout occurs, only the current write request is abandoned; the driver starts anew on the next one even though output is still blocked.

If the write timeout policy is for indefinite waiting (Mode command *TO=off*) the driver will wait forever for output to be unblocked. Therefore the process that issued the *DosWrite* may be delayed indefinitely. When an indefinite wait is used, it is a good idea to use *DosWriteAsync* for serial output; that way the main thread retains control while output is blocked.

References: *DosWrite* 8.11.4; *DosWriteAsync* 8.11.3

14.3.3.1.2 Concurrent Writes

It is possible for multiple, asynchronous threads to write to the same serial port. This could happen if two or more processes opened files to the same serial port.

or if two or more threads within one process issued *DosWrite* calls, or if a single thread issued more than one *DosWriteAsync* call. Another possibility is that one thread might work on the port with IOCTL calls while another thread's output is being written.

In general there is no way to predict which *DosWrite* request would reach the device driver first. Even multiple *DosWriteAsync* calls from a single thread might not always arrive in the order in which they were started.

The driver works on one request at a time, but the requests from different threads will be mixed in the output stream. Ordinarily only one process will open a file to the port and only one thread within the process will be doing *DosWrite* or *DosDevIOCTL* calls. When that is the case, output will be transmitted in the order of the *DosWrite* calls.

References: *DosWrite* 8.11.4; *DosWriteAsync* 8.11.5

14.3.3.2 Reading

Input from a serial port is the same as input from any file: a process calls *DosRead* (or *DosReadAsync*) with a handle, a buffer, and a requested number of bytes. OS/2 translates this into a call on the serial driver with a request to read. The driver queues read requests in the order it receives them. It works on one read request at a time. In all cases it begins by copying bytes from its input queue to the caller's buffer. If there are enough queued bytes to satisfy the request, the input request is marked as finished and returned to OS/2. The *DosRead* call ends with a received count equal to the requested count.

References: *DosRead* 8.11.2; *DosReadAsync* 8.11.5

14.3.3.2.1 Read Timeouts

When there is not enough queued data to fulfill an input request, the driver consults the read timeout policy noted in the DCB. Ordinarily this will be *wait for timeout or full count*. Under this policy the driver waits for more input to arrive. If a character arrives before the read timeout interval (a number in the DCB, initially 60 seconds) has expired, the driver puts the character in the caller's buffer and waits again with a new interval.

It continues to wait and to store arriving characters in the buffer until it has fulfilled the count of requested bytes, or until it has to wait so long that the read timeout interval expires. If it fulfills the request, the count of received bytes returned by *DosRead* will equal the count of requested bytes. If a timeout occurred, the received count will be less than requested and might be zero.

Another read timeout policy is *wait for timeout or nonzero count.* Under this policy, the input request is finished as soon as one or more bytes have been transferred. If there are any queued bytes at all, the request takes them and ends at once without waiting for more. The received count will reflect the number of bytes that were queued when work began on the input request.

Only if there were no queued bytes at all will the driver wait for input, and then it waits only once. If the read timeout expires the request ends; or as soon as any characters arrive and are queued, they are used to satisfy the input request. Usually only one character will arrive, but at high transmission rates in a busy system, more than one might be queued before the driver gets around to finishing the job. Under this policy the received count will usually be less than the requested count, and a received count of zero indicates a timeout.

A third read timeout policy is *no waiting.* Under this rule the driver never delays an input request. It fulfills the request with however many characters are in the input queue, up to the requested number, and declares the request finished. Under this policy the received count will usually be less than the requested count, and received counts of zero will be frequent.

14.3.3.2.2 Concurrent Reads

It is possible for multiple, asynchronous threads to be reading from the same serial port. This could happen if two or more processes opened files to the same serial port, or if two or more threads within one process issued *DosRead* calls, or if one thread issued two or more *DosReadAsync* calls. Another possibility is that one thread might work on the port with IOCTL calls while another thread's input request was being processed.

In general there is no way to predict which *DosRead* request would reach the device driver first. Even multiple *DosReadAsync* calls might not always arrive in the order in which they were started. The driver works on one request at a time, but requests from different threads will be mixed in its input queue, and the received data will be distributed among them.

Ordinarily only one process will open a file to the port and only one thread within the process will be doing *DosRead* or *DosDevIOCTL* calls. When that is the case, all input will be read (barring overruns) in the order in which it arrived.

References: *DosRead* 8.11.2; *DosReadAsync* 8.11.5

14.3.3.2.3 Read Errors

A variety of errors can affect the receipt of a character. The hardware may note a parity error when parity is enabled. There is also a chance of what is called a *framing error*, in which the hardware sees, or thinks it sees, the start of a character but does not find a stop bit at the end of the character. Constant framing errors result when the sending and receiving systems aren't using the same character size; intermittent framing errors can be caused by noise on the line.

A break signal on the input line is also an error of a kind; it usually causes a parity error and a framing error, as well as being recognized as a break.

Each of these events is unscheduled. They are significant to the program reading the data, but there is no way to predict when they are going to happen. Worse, there is no reliable way for the program to find out when they *did* happen. It can use IOCTL calls to poll the device driver, and so find out that an error has happened. But in a multiprogramming system there is always uncertainty: the program can find out that a break signal or parity error has been detected, but it cannot find out precisely *when*; that is, where in relation to the block of input data being read.

The solution is to have the device driver convert an error into a data character and enter it into the input queue at the point where it was detected. This is a perfect solution provided that the program can spare one or two character values for this purpose. The device driver can be told to use a policy of substituting a particular byte value where an error is detected, and can be told to use the same or another byte value to represent a break signal.

There is a sad irony here. One of the ASCII control characters, *SUB*, was designed for *precisely* this purpose: to stand in place of any other character that was received in error. Unfortunately, long ago and in a distant operating system, some short-sighted programmer—probably strung out on caffeine at three in the morning—was captivated by the notion of representing *the end* with a control-Z, and chose to use *SUB* as a logical end of file marker. This, despite the plain fact that there were two other little-used ASCII characters that already carried that

meaning (End Medium and File Separator). What's more remarkable is that this stupidity has been propagated down and down the generations of operating systems, and we are *still* stuck with it today in OS/2. So if you choose *SUB* to stand for an error character, and if an error character gets into a disk file or a pipe, it may be mistaken as marking end of file. Sad, sad, sad.

14.3.4 Serial Driver IOCTL Usage

The serial port device driver responds to a number of *DosDevIOCTL* calls, using category code 1. The function codes supported are summarized in Table 14.3.

All *DosDevIOCTL* calls take five parameters: the category code (here always 1), the function code (from Table 14.3), a handle that implicitly specifies the device driver, and finally the addresses of a parameter packet and a data packet. The two codes and two addresses are passed to the serial driver, which interprets them, performs a function, and returns an error code.

If the serial driver was able to perform the function, it returns an error code of zero (and possible some data in the data packet). If it cannot do the operation or does not like one of the parameters, it returns error code 31, which means "general failure."

When either the parameter packet or the data packet is not relevant to a function, the serial driver insists that the irrelevant address be passed as a null pointer, that is, as a doubleword of zero on the stack. This is indicated in the third and fourth columns of Table 14.3 by the word *null*. For example, suppose that a program has declared a handle for the serial port named *Port* and an integer variable named *BitRate*. The *DosDevIOCTL* call to query the transmission rate (see the first line of Table 14.3) would be coded this way in a C program:

```
rc = DosDevIOCTL( &BitRate          /* data packet */
                 ,(void far *)0L    /* parm packet*/
                 ,0x01              /* category */
                 ,0x62              /* function */
                 ,Port);            /* handle */
```

The expression *(void far *)0L* is a magic incantation that says "doubleword of zero treated as a far pointer." The same idea is only a little harder to express in Pascal. The include files in the Disk Appendix of this book declare a null far

Table 14.3
Summary of Generic IOCTL Functions in Category 1

Function Code (hex)	Operation done	Parameter Packet	Data Packet
61	Query bit-rate	null	integer
41	Set bit-rate	integer	null
62	Query line protocol	null	Figure 14.2
42	Set line protocol	Figure 14.2	null
73	Query Device Control Block	null	Figure 14.3
53	Set Device Control Block	Figure 14.3	null
68	Query input-queue sizes	null	2 words
69	Query transmit-queue sizes	null	2 words
64	Query handshake status	null	Figure 14.4
65	Query transmit status	null	Figure 14.5
67	Query input signal lines	null	Figure 14.6
6d	Query error events	null	Figure 14.7
72	Query line status	null	Figure 14.8
4B	Set break line condition on	null	Figure 14.7
45	Set break line condition off	null	Figure 14.7
66	Query RTS, DTS lines	null	Figure 14.9
46	Set RTS, DTS lines	Figure 14.9	Figure 14.7
44	Send immediate byte	byte	null
47	Pretend XOFF received	null	null
48	Pretend XON received	null	null

A parameter or data packet of "null" must be specified as a null address (doubleword of zero) in the DosDevIOCTL call.

pointer under the name *NilADS*, this way:

```
const NilADS = ADS(0,0);
```

Using it, the above IOCTL could be coded in a Pascal program like this:

```
rc := DosDevIOCTL( ADS BitRate (* data packet *)
                  ,NilADS      (* parm packet *)
                  ,16#01       (* category *)
                  ,16#62       (* function *)
                  ,Port);      (* handle *)
```

Following either call, if the variable *rc* contained zero, the variable *BitRate* would contain the current transmission rate in bits per second.

14.3.4.1 Querying and Setting Line Protocol

The RS-232 line protocol comprises the transmission rate, character size, parity form and number of stop bits—the positional parameters of the Mode command or of BASIC's *OPEN COM...* statement. There are two ways of setting the line protocol from within a program.

Perhaps the best way is to invoke the Mode command as a child process! Prepare the mode command as a string prefixed with the characters */c*, for instance */c mode com1:4800,e,8,2,to=on,idsr=off*. Use the *DosScanEnv* call to locate the *COMSPEC=* string in the environment; its value is the pathname of the command shell. Use these strings with *DosExecPgm* to invoke the command shell and have it execute a mode command for you. While only a little simpler than using IOCTLs, this has the advantages of clarity and portability, and it can be tested easily.

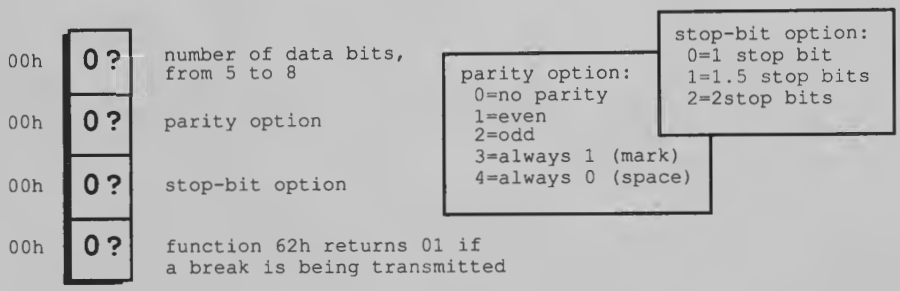
Notice that this scheme can only work because the serial driver permits more than one file open to a port. The Mode command uses *DosDevIOCTL*, and in order to do that it has to open a file to the serial port. Since more than one open is allowed, your program can open the file and then later call the Mode command; its open and subsequent close will not affect the operation of the file. However, the Mode command opens the file for read access; if your program has the serial port open with a share option that denies readers, the Mode command will fail.

Alas, most programmers would rather do it themselves. As noted in Table 14.3, the transmission rate is queried with IOCTL function 61h. The value returned is the transmission rate in bits per second. The complementary function 41h sets the transmission rate; the parameter packet must be a word giving the desired rate which must be one of the supported rates (110, 150, and successive doubles of 150).

The remaining components of the protocol are queried with IOCTL function 62h. It returns the four-byte structure diagrammed in Figure 14.2 at the data packet address. Function 42h sets the protocol items, taking the same structure as its parameter packet (it ignores the fourth byte).

References: *DosScanEnv* 5.1.5.2.1; *DosExecPgm* 6.1.2

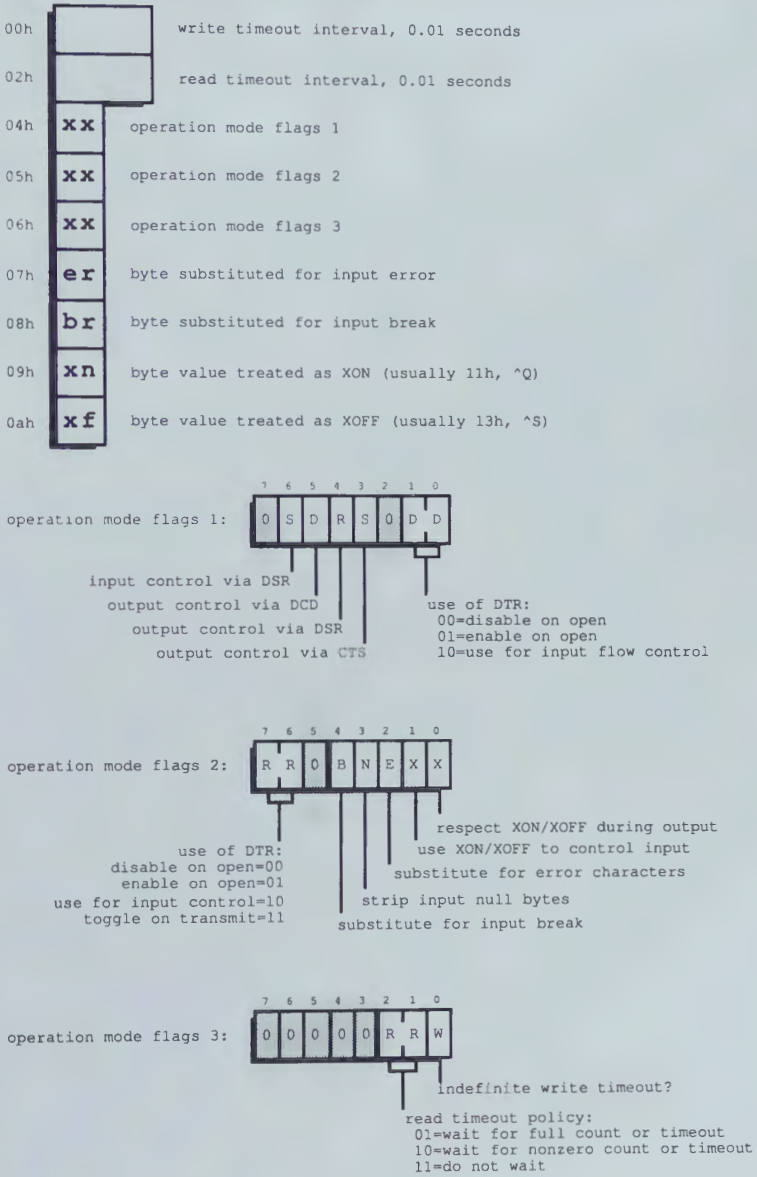
Figure 14.2
Line Protocol Values for IOCTL 1:62h and 1:42h



14.3.4.2 Querying and Setting the DCB

IOCTL function 73h returns the contents of the Device Control Block at the data packet address. Device Control Block is the unimaginative name that Microsoft gives to the structure diagrammed in Figure 14.3; I think a better name would be Device Policy List, because it specifies the many policies that apply to serial input and output.

Figure 14.3
Device Control Block Used with IOCTL 1:73h and 1:53h



Function 53h is the complement to 73h; it takes a DCB structure as its parameter packet and, if all the fields are valid, installs it. All policies go into effect immediately; if input or output is in progress at that time the effects are unpredictable!

14.3.4.2.1 Setting Timeout Policy

The first two words in the DCB are the timeout intervals for write and read respectively. The timeout values are given in units of 0.01 seconds, not in milliseconds like most intervals in OS/2. The values are initialized to 6000, meaning one minute. The longest timeout interval that can be specified is 655 seconds, just short of eleven minutes, but more often a program would want to set a shorter timeout. For instance, if a remote system should block output for longer than ten seconds, it is probably in some kind of trouble.

After the the timeout intervals have been set, the actual timeout policy is set in the third operation mode flag byte (DCB offset 06h as shown in Figure 14.3).

There is only one choice for the write timeout. The bit may be set to zero to request timeouts during write (*Mode TO=on*). Or it may be set to 1 to request indefinite waits during write (*Mode TO=off*). Indefinite waits make sense with some devices (a program writing to a printer that runs out of paper really ought to wait forever, or at least until the operator returns from lunch) but not all.

There are three choices for the read timeout policy; their implications were discussed above in regard to input processing. This policy is initially to wait for full count or timeout; it can't be set by the Mode command.

14.3.4.2.2 Substitution Policies

The device driver can be asked to substitute any byte value in place of a character that is received in error (received with a parity error or framing error). It can also be asked to insert any byte value in the input stream when a break is detected. When these substitutions are done, the program can tell exactly where in the input stream the error or break occurred. The cost is that one or two byte values have to be reserved for the purpose; if the same byte values might arrive as data, they will be ambiguous.

The byte values to be used for substitution are set in the DCB at offsets 07h and 08h. These fields are cleared to zero on a first open of the port, so they must always be reset following an open.

The substitution features are enabled by two bits in the second operation mode flag byte (offset 05h in the DCB; see Figure 14.3).

14.3.4.2.3 Setting Policy on Null Data

Null bytes (ones with a value of zero) may represent meaningful data, but some hardware can generate meaningless nulls. If nulls are always meaningless the device driver can be asked to discard them. This policy is set with a bit in the second operation mode flag byte (offset 05h in the DCB).

14.3.4.2.4 Setting DTR Policy

The serial driver's use of the DTR signal is set by a two-bit number in the first operation mode flag byte (offset 04h in the DCB; see Figure 14.3 [DEV.DCB]). There are only three possibilities.

The policy set initially (and by *Mode DTR=on*) is *enable on open*; DTR is set to a true state on a first open of the port and to a false state on a last close. This simulates the operation of a terminal; DTR says *powered on and ready* so long as any program has a file open to the port. The opposite policy is *disable on open* (*Mode DTR=off*); DTR is left in the false state.

When either of these policies is in effect, the program may use IOCTL function 46h to set specific values on the DTR line. If the program is going to control the state of DTR this way, *disable on open* is probably the best policy.

The third policy is to use DTR for input flow control (*Mode DTR=hs*). Under this policy the device driver will set DTR true on a first open and leave it true so long as the input queue length is below a threshold value. When the queue exceeds the threshold, the driver will swing DTR false to discourage further input. Before using this policy you must be very sure that the connected device expects DTR to be used this way (or that DTR is wired to some other other signal such as CTS). A modem, for instance, that sees DTR go false is likely to drop the telephone connection.

When this last policy is in effect, the program may query the state of DTR with IOCTL function 66h, but function 46h will not be allowed to change it.

14.3.4.2.5 Setting RTS Policy

The serial driver's use of the RTS signal is set by a two-bit number in the second operation mode flag byte (offset 05h in the DCB; see Figure 14.3). There are four possibilities.

The policy set initially (and by *Mode RTS=on*) is *enable on open*; RTS is set to a true state on a first open of the port and to a false state on a last close. This simulates the operation of a terminal; RTS is true so long as any program has a file open to the port. The opposite policy is *disable on open* (*Mode RTS=off*); the signal is left in the false state.

When either of these policies is in effect, the program may use IOCTL function 46h to set specific values on the RTS line. If the program is going to control the state of RTS this way, *disable on open* is probably the best policy.

The third policy is to use RTS for input flow control (*Mode RTS=hs*). Under this policy the device driver will set RTS true on a first open and leave it true so long as the input queue length is below a threshold value. When the queue length exceeds the threshold, the driver will swing RTS false to discourage further input. Before using this policy you must be very sure that the connected device expects RTS to be used in this highly nonstandard way (or that RTS is wired to some other other signal such as CTS).

The fourth policy is *toggle on transmit* (*Mode RTS=tog*); it requests the device driver to signal on RTS whenever it has data to transmit. This includes any time there is data in the transmit queue, any time there is an immediate byte being sent with IOCTL function 44h, any time the driver needs to send an *XOFF* or *XON* for flow control, and any time a break is to be signalled. The device driver sets RTS false whenever it has no data to transmit, and whenever output has been blocked by receipt of an *XOFF*. This policy implements the original intent of the RTS signal in a strict sense; it may be needed by some types of hardware.

When these last two policies are in effect, the program may query the state of RTS using IOCTL function 66h, but IOCTL function 46h will not be allowed to change it.

14.3.4.2.6 Setting Input Flow Controls

It's just about impossible for the application to implement a flow control policy at the level of characters (higher levels, flow control at the level of logical records or blocks, is certainly in the scope of the application). Character-level input happens asynchronously and the application cannot be sure it will be allowed to use the CPU in time to respond to an excess. Only the device driver can do this because only it gets control during interrupts.

Four different input policies can be set independently in the DCB. Two are mentioned above: use of either DTR or RTS (or both, for heaven's sake) for hardware flow control.

A third policy is not strictly for flow control: the device driver can be asked to ignore input characters unless the DSR signal is true. Some devices generate good input regardless of the state of DSR, but most often input when DSR is false is garbage. The input-DSR policy is set in the first operation mode flag byte (DCB offset 04h); a 1-bit asks the driver to use DSR as an input gate (*Mode IDSR=on*) and a zero says to ignore DSR (*Mode IDSR=off*).

All these are local policies that operate between the computer and the attached device. The only input flow control policy that can operate over a telephone line to a remote computer is one that uses *XOFF* and *XON* (or whatever byte values are set in the DCB fields at offsets 09h and 0ah). Input flow control with *XOFF* is enabled by bit 1 in the second mode flag byte (DCB offset 05h). When it is enabled, the device driver will *transmit* an *XOFF* whenever it *receives* a character that brings its input queue past a threshold value. When later a *DosRead* brings the input queue below the threshold, the device driver will transmit an *XON* to resume input.

This policy is of course useless when the remote system does not respect *XON* and *XOFF* for output flow control. It is also useless when *XON* and *XOFF* are unavailable for this function; that is, when 8-bit data bytes are being sent between computers and all byte values are significant.

This policy does *not* correspond to the Mode parameter *XON=on;* input flow control cannot be set by the Mode command.

14.3.4.2.7 Setting Output Flow Controls

Just as with input flow control, it's just about impossible for an application to implement output flow control at the character level. Flow control at the level of logical records or blocks is definitely up to the application, but only the device driver can respond quickly enough to the requests of the attached device or remote computer for a pause in the output stream.

Four output flow control policies can be used in any combination. Three of them are set by bits in the first mode flag byte (DCB offset 04h). The device driver can be asked to check for a true state of the CTS, DSR, and/or DCD signals before transmitting. Any of these signals might be taken as blocking output, depending on the attached device. DSR false would normally indicate that there was no attached device or it was powered off. CTS false would normally indicate that it was refusing permission to transmit, rejecting the signal on the RTS line. These two policies are initially on and can be changed with the Mode command (*ODSR=on* and *OCTS=on*).

The third policy, testing DCD, is initially off and can't be set by the Mode command. Most serial devices set no meaningful value on DCD. A modem would normally use DCD to indicate that there was no active connection to the telephone line but, as mentioned above, output to a smart modem might be the only way of getting it to make a connection. However, DCD or any of these signals might be patched to any other signal line; all manner of strange interfaces are perpetrated with a breakout box. You simply must know how the attached device is designed and what it expects.

These three are local policies that affect transmission to the attached device. Only a policy based on sending data characters can be effective across a telephone link. Bit zero of the second mode flag byte asks the device driver to watch for and respect the appearance of *XOFF* and *XON* (or whatever byte values are in the DCB at offsets 09h and 0ah) during output. When it sees an *XOFF* it blocks further output until it sees an *XON*.

This policy is controlled by the Mode command's parameter *XON=on*. It should only be enabled when the connected system requires it and when the *XON* and *XOFF* (or other byte values as set in the DCB) have been reserved for flow control and only flow control. The reason is that the device driver is very faithful: when flow control is enabled and an *XOFF* appears in the input stream then (1) the *XOFF* is not placed in the input queue; (2) further input is ignored until an *XON* is received; and (3) output is blocked until the *XON* is received. In short, the device driver shuts down both sides of the interface until the remote system

unblocks it. If the *XOFF* was intended as data, the application will be frozen until the remote system sends an accidental *XON* value—if it ever does.

14.3.4.3 Querying Queue Sizes

The IOCTL functions 68h and 69h (see Table 14.3) get information on the status of the serial driver's input and transmit queues respectively. For either function, the data packet is a pair of words. The first word receives the number of characters presently in the queue, while the second receives the size of the queue.

The sizes of the queues must be interpreted carefully. We are warned that they may change from one release of the system to the next, and might even change from one open of a port to the next open.

It does a program little good to know the specific number of characters in the transmit queue. It is useful to know when there are no characters queued, because when the queue is empty, and no write requests are queued, and the transmit shift register is empty, then all output is complete and it is safe to close the file. However, these facts can all be discovered with IOCTL 65h, discussed below.

A program may poll the serial driver with IOCTL 68h to find out when there is some queued input. The count of queued characters could be used as the requested count in *DosRead*, so as to ensure that the *DosRead* would complete at once regardless of the read timeout policy in effect (assuming, of course, that no other thread was reading at the same time). If it is sufficient to know that at least one input byte has been queued, that fact is available from IOCTL 72h.

14.3.4.4 Querying Port Status

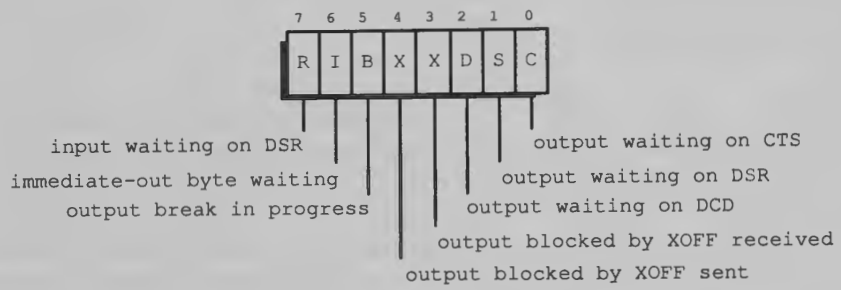
The serial driver will return a frightening amount of status information in response to IOCTL functions 64h, 65h, 67h, 6dh, and 72h, as summarized in table 14.3. Most applications will need little of this information but you should be aware that it's available.

14.3.4.4.1 Handshake Status from IOCTL 64h

Function 64h returns the byte diagrammed in figure 14.4 in its data packet. These bits reflect all the things that might block output: false states of the CTS, DSR, or DCD lines, or an *XOFF* received from the remote system, or the fact of

having sent an *XOFF* to request a pause in the input stream. That event blocks output because (the driver assumes) the remote system will ignore all subsequent input until it sees the *XON*.

Figure 14.4
Handshake-Status Bits Returned by IOCTL 1:64h



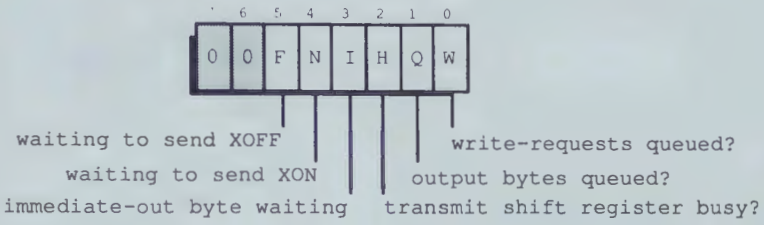
This byte also reflects three miscellaneous states: of a break being sent (IOCTL 4bh); of an immediate output byte waiting to be sent (IOCTL 44h); and of an input request being blocked by a false state of DSR.

14.3.4.4.2 Transmit Status From IOCTL 65h

Function 65h returns the byte shown in Figure 14.5 as its data packet. These flags reflect the state of output requested. One bit shows whether there are any bytes in the transmit queue (and thus can take the place of function 69h for most purposes); another shows whether any more write requests are queued but not yet being worked on.

When all of the bits of this byte are zero, there is absolutely no output remaining to be done. This is the fundamental test as to whether the program may close the port without danger of losing any output. Of course, if the handshake status that returned function 64h shows that output is blocked, there isn't much point in waiting for this byte to become zero; it can't.

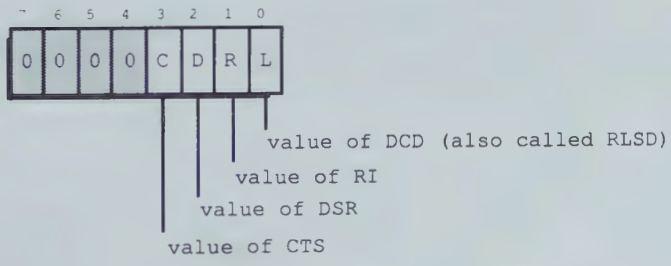
Figure 14.5
Transmit Status Bits Returned by IOCTL 1:65h



14.3.4.4.3 Input RS-232 Signals From IOCTL 67h

Function 67h returns the byte shown in Figure 14.6 as its data packet. This byte reflects the state of the RS-232 signals that are output by a DCE device and input to a DTE.

Figure 14.6
Input-Signal Bits Returned by IOCTL 1:67h

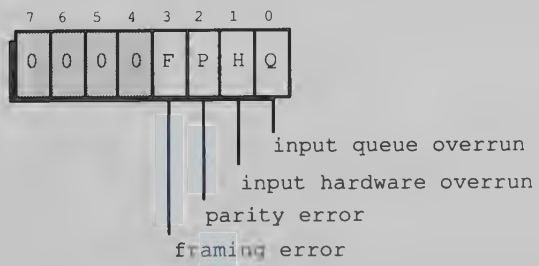


Three of these signals can be used for output flow control. Ring Indicator, if it is implemented by a modem, should rise and fall with the presence of ring tone on the telephone line—but it has been used for many other purposes by custom hardware.

14.3.4.4.4 Error Events From IOCTL 6dh

Function 6dh returns the byte shown in Figure 14.7 in its data packet. These bits reflect the occurrence of input errors.

Figure 14.7
Line-Error Bits Returned by IOCTL 1:6dh



The bits of the error status byte are set to zero on a first open. Then they accumulate until they are queried with function 6dh, after which the driver sets them to zero again. If it could poll often enough, the program could see every error condition. However, no program in an OS/2 system can be sure of polling that fast. A program simply cannot know how many other processes are in the system or how often it will get a turn to run. Any number of line errors might occur between two calls of this function.

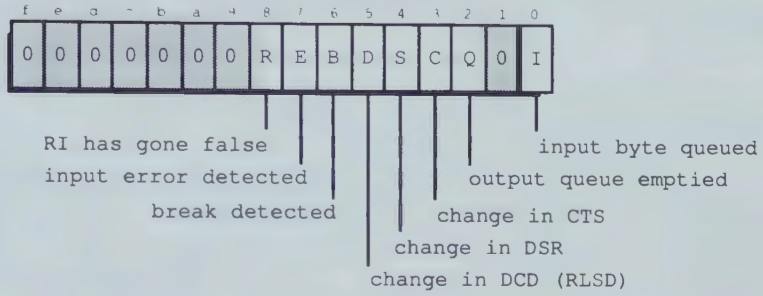
14.3.4.4.5 Transient Events From IOCTL 72h

Function 72h returns the word (notice it's a word, not a byte as returned by the preceding functions) shown in Figure 14.8 as its data packet. These bits reflect transient events that occur in the normal course of operations on the port. All these bits are cleared on a first open of the port, and again after every query.

Bit 0 is set true whenever an input byte is taken from the hardware and set in the input queue. A program can't be sure of counting input bytes by polling this bit, but a 1-bit here indicates that some input has been received.

Bit 2 is set whenever the output queue is emptied, but it does not mean that no more output remains. Use function 65h for that test.

Figure 14.8
Line-Event Bits Returned by IOCTL 1:72h



A one in bit 6 means that the hardware has recognized the break condition (continuous spacing on the input data line). There may also be an input error associated with a break. An input error is signalled in bit 7; use function 6dh to find out what kind it is.

14.3.4.5 Sending a Break Signal

An output break signal is created by setting the output data line to the space condition and holding it there for several character times, usually for 350 to 500 milliseconds. A break is really a forced framing error; its length distinguishes it from ordinary line noise.

The break signal is recognized by only some remote computers (the GEnie and Lockheed DIALOG information services support it; most other services and bulletin boards do not). It is almost never used with devices other than modems.

A program begins sending a break signal with IOCTL function 4bh. The driver does not check to see if a character is being transmitted at that time, and if one is, it will be spoiled. However, the driver does block further output (as will be reflected in the handshake status returned by IOCTL 64h) until the break is turned off.

After a suitable delay (perhaps with a *DosSleep(350)* call) the program ends the break signal with IOCTL function 45h. The transmit data line is returned to the marking state and the output block is lifted.

Both these functions return the same data packet as function 6dh, as shown in Figure 14.7. The serial driver doesn't object to multiple, overlapping calls for break signals. The first function 4bh call starts the break; the first function 45h call that follows ends it.

14.3.4.6 Custom Interface Control

The remaining IOCTL functions allow a program to use the serial port interface in ways that have not been anticipated by the device driver. In particular, a program may try to implement a private flow-control policy.

14.3.4.6.1 Control of RTS and DTR

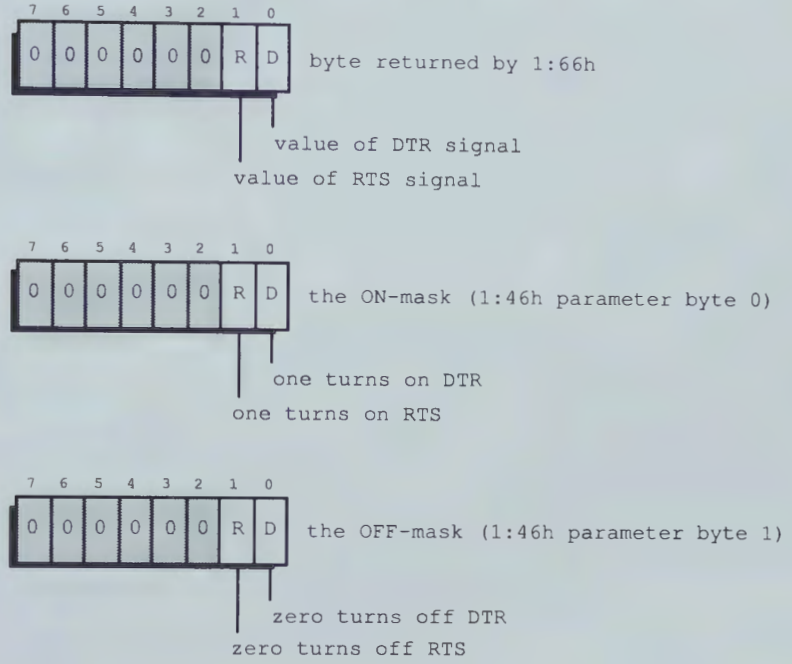
The state of the RTS and DTR signals may be queried with a call to IOCTL function 66h. It returns a byte as shown in Figure 14.9 in its data packet. This simply reflects DTR in bit zero and RTS in bit 1.

When the policies for DTR and RTS (as set in the DCB by IOCTL 53h) are either *enable on open* or *disable on open*, the program may command changes in their values with function 46h. This function takes a parameter packet of two bytes. The first byte is the on-mask, and the second is the off-mask, as shown in Figure 14.9. The off-mask is applied first, then the on-mask, so that if the masks turn the same bit off and on, it will finally be left on. Notice (in table 14.3) that this function returns a data packet, unlike many control functions that do not.

14.3.4.6.2 Sending Immediate Data

The program may send a single byte with high priority using IOCTL function 44h. The immediate byte does not pass through the transmit queue; there is a special one-character buffer for it. The device driver can hold just one immediate output byte at a time; the status byte returned by function 64h (see Figure 14.4) shows whether one is already waiting to be sent. If one is already pending, another call to function 44h will be rejected with an error code.

Figure 14.9
Output-Signal Masks for IOCTL 1:46h and 1:66h



Immediate output (and all other output) can be blocked by hardware signals CTS, DSR, and DCD (use function 64h to see if this is the case). When output is not blocked by those signals, the immediate output byte will be sent next time the transmit hardware is ready. It will not be blocked by either an *XOFF* received or an *XOFF* sent. However, any data sent when output is blocked by logical flow control is likely to be ignored or misunderstood.

If the device driver needs to send an *XOFF* or an *XON* (see the status returned by function 65h, Figure 14.5 to see if this is the case) the *XOFF* or *XON* byte may or may not precede the immediate byte in the output stream.

14.3.4.6.3 Controlling Output-Block Status

The program can make the device driver behave as if it had received an *XOFF* or *XON* character. It might do this in order to implement flow control based on some other kind of input.

Function 47h tells the device driver to behave as if it had received an *XOFF* byte with *XOFF* flow control enabled. It will set an output block (reflected in the byte returned by function 64h, Figure 14.4) and stop transmitting ordinary data. It will still transmit an immediate output byte requested through function 44h. If input flow control using *XOFF* is enabled, it may transmit *XOFF* or *XON* bytes to control input flow.

If output flow control with *XOFF* is enabled, receipt of an *XON* character will end the output block as usual.

Function 48h tells the device driver to remove a logical output block, regardless of whether it was imposed by function 47h or by receipt of an *XOFF* character.

15

Whole-Disk Operations

Ordinarily programs only access disks as files. The program knows only the file name and its contents; all the details of how the disk is organized to create a system of files and directories is left up to OS/2.

Only rarely is there a good reason for operating on a disk as a device, reading sectors of disk data without regard for file boundaries. Rarer still is there a good reason to write a physical disk sector. These things can be done; in part they require the use of IOCTL calls to the disk device driver. Some of the available IOCTL functions are summarized in Table 15.1

Table 15.1
Selected Generic IOCTL Functions in Category 8

Function Code (hex)	Operation done	Parameter Packet	Data Packet
00	Lock disk volume	byte of 00h	zero doubleword
01	Unlock disk volume	byte of 00h	zero doubleword
02	Redetermine volume format	byte of 00h	zero doubleword
20	Query if fixed disk	byte of 00h	byte 0 or 1
63	Query disk info (BPB)	byte 0 or 1	Figure 15.1

15.1 Opening a Disk as a File

The *DASDOpen* option of the *DosOpen* Mode parameter selects a unique and very powerful ability: the ability to treat the entire surface of a disk volume as a single file, without regard for the organization of the file system. (*DASD* stands for Direct Access Storage Device; that is, disk. It's a venerable acronym that was applied to the very earliest disk devices sold by IBM.)

For example, if *DosOpen* is given a path of merely *A:* with the *DASDOpen* option, appropriate share and access options, and a Flag word of *fail new, open existing*, it returns a handle that represents the entire surface of the diskette in the A drive. An offset of zero in this file represents the contents of sector 1 on track zero; an offset of 512 is sector 2, and so forth. *DosRead* and *DosWrite* work normally in this super-file.

So long as a program only reads, no serious harm can come from a *DASDOpen* file. Any enterprising hacker can cobble up a useful disk browsing utility based on *DASDOpen* in next to no time.

> **Reference:** *DosOpen* mode word 8.7.3; access options 8.7.3.1; share options 8.7.3.2; *DosRead* and *DosWrite* 8.11

15.2 Locking a Disk

Obviously a great deal of harm could come from careless writing to a *DASDOpen* file. The disk directory or File Allocation Table could easily be trashed. More constructively, a *DASDOpen* output file could be the basis of an unerase and disk patch utility.

However, a program that plans to write to a *DASDOpen* file ought to make sure that it has exclusive use of the disk in question. Exclusive use is also necessary for some applications that only read; for instance a fast backup utility needs to know that the disk isn't changing while it is being copied.

The Share option of *DosOpen* is *not* effective for this. The Share option of a *DASDOpen* is tested only against other *DASDOpens*, not against ordinary file opens. For example, if one program opens *A:* with Share option *deny writers*, a second program that tries to open *A:* for write access will be denied. However, specific files may be open on that disk with any kind of Share options, and the *DASDOpen* will succeed. Nor can a *DASDOpen* file be locked using *DosFileLocks*.

How, then, does a program get exclusive use of a disk? Through one of the generic *DosDevIOCtl* calls. As summarized in Table 15.1, the handle returned from a *DASDOpen* may be used in a *DosDevIOCtl* call for category 8, function 0, with a data packet of a doubleword of zero and parameter packet of a byte of zero. This incantation requests the disk device driver to lock the drive to access from all other processes.

An error will be returned if any other process has an active file or *DASDOpen* on the drive. If the call succeeds, the process has exclusive use of the entire drive and will keep it until it calls for generic IOCtl category 8, function 1 with the same data and parameter areas.

It is never possible to get exclusive use a fixed disk that is the OS/2 boot drive. The reason is that the OS/2 kernel holds open the dynamic link libraries it uses (*DOSCALLS.DLL*, *VIOCALLS.DLL*, etc.). Since it never closes them, the disk driver will never be able to give another process exclusive use of the boot drive. If it's necessary to format the fixed disk that's normally the boot drive, the system will have to be booted from another drive.

For a similar reason it is never possible to get exclusive use of a fixed disk from which a program itself was loaded. A loaded program's executable file is kept open so long as the program is in storage, and that blocks exclusive use of the disk.

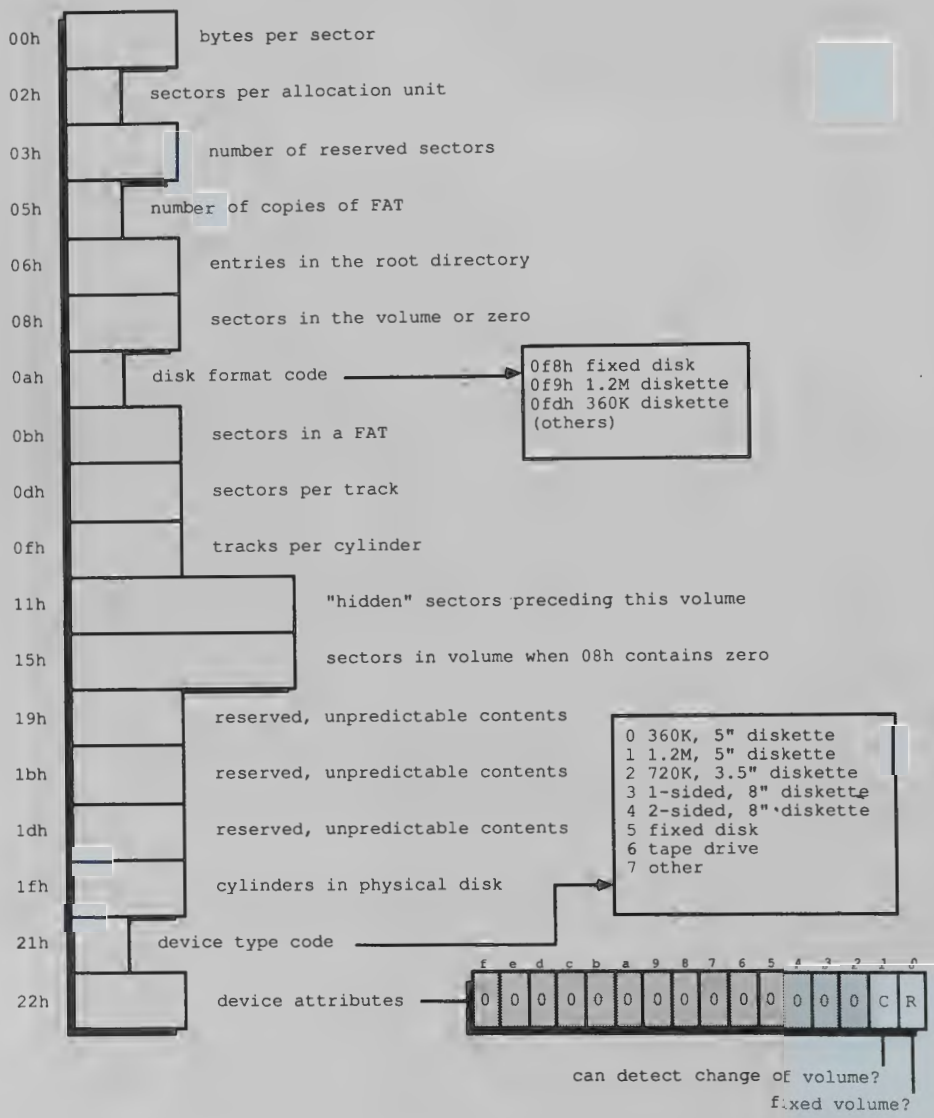
> **References:** *DosOpen* share options 8.7.3.2; *DosFileLocks* 8.11.8; *DosDevIOCTL* coding 14.3.4

15.3 Querying Disk Information

IOCTL category 8, function 63h, returns a wealth of information about the disk volume associated with a *DASDOpen* handle (actually it will accept any handle for a disk file).

When disk volumes are removable, the OS/2 disk device driver maintains two sets of device information. One set describes the usual, expected disk format. The other set describes the volume presently mounted in the drive. The parameter packet for IOCTL category 8, function 63h specifies which one of these sets of data the program wants. A parameter byte of zero asks for the expected set; a byte of 1 requests the data for the current volume. The information is returned in the data structure shown in Figure 15.1.

Figure 15.1
Disk Volume Information Rteurned by IOCTL 8:63h



The first thing to note about this structure is that several of its fields are aligned at odd-numbered offsets. If you declare the structure to most Pascal or C compilers they will insert padding bytes to bring the words and doublewords to appropriate boundaries, and your declaration will not match the reality. When using Microsoft Pascal you will have to declare explicit field offsets for every field. For example:

```
type DiskInfo = record
bytes_per_sector [00] : word;
sectors_per_unit [02] : byte;
reserved_sectors [03] : word
```

and so on. Using Microsoft or IBM C you can specify the option */Zp* to force the compiler to pack structure members to byte boundaries; however this will affect all structures declared in that source module. Some C compilers support a pragma that allows you to pack single structure declarations.

The second thing to note is that Microsoft and IBM have promised that the file system supported by MS-DOS and OS/2 will soon be upgraded to support larger volume sizes. Therefore you should expect that some of this information may change in future releases.

Finally, realize that the structure of Figure 15.1 describes a logical disk volume, and not necessarily an entire physical disk. On many fixed disks and all diskettes, a logical volume does correspond to a physical volume. But some fixed disks are divided into two or more partitions (using the Fdisk utility, for example). A partition of a fixed disk is a complete disk volume with its own drive letter.

If you have had experience with the details of the MS-DOS file system, you may recognize the bulk of figure 15.1 as a *BIOS Parameter Block*, or BPB. It gives a good picture of the physical and logical layout of the volume. The fields that depict the physical layout include:

- offset 00h, bytes per sector
- offset 0dh, sectors per track
- offset 0fh, tracks per cylinder
- offset 1fh, number of cylinders.

From these the total number of sectors should be easy to calculate—but it is not. The total-sector count may not agree with a simple multiplication of these fields.

There is the matter of physical disk versus logical volume. The number of cylinders given at offset 1fh apparently describes the physical disk. The number of physical sectors that are reserved to preceding partitions of the disk is given in the 32-bit number at offset 11h. These sectors are hidden in the sense that they are to be ignored when working with this logical volume. In a sense they give the starting offset of the volume within the disk; at any rate they are not counted in the total sectors for the volume.

There is also the chance that the total number of sectors in the volume might exceed the capacity of a 16-bit word. If it does, the field at offset 08h will contain zero and the actual number of sectors will be in the 32-bit number at offset 15h.

The six bytes at offset 19h are undocumented. I have been assured by Microsoft technical people that these are reserved and have no assigned meaning. Unlike most reserved fields, they are not zero; their content varies with the drive.

15.4 Handling Removable Volumes

Disk volumes that can be removed (all diskettes and some unusual hard disks) are fundamentally different from ones that are fixed in their drives. It may be important for a program to know whether a disk volume is fixed or removable. IOCTL category 8, function 20h (see Table 15.1), when applied to a *DASDOpen* handle, will return as its data packet a byte containing zero if the associated volume is removable and 1 if it is a fixed drive. (Actually, a *DASDOpen* handle isn't required; the handle may be that of any disk file.)

The same information is available from category 8, function 63h, but there it is embedded in a much larger structure that the program may not need.

If a volume is removable, and if the program directs the user to change disks, it may be useful to alert the device driver to the change. The OS/2 kernel may catch on to the change next time it opens a file on the disk; but the program might not be using the OS/2 file system. IOCTL category 8, function 02h (see table 15.1) tells the device driver to sense the disk format again.

16

Device Monitors

Some device drivers support *device monitors*, programs that are allowed to inspect, censor, and modify the flow of data that passes between the device driver and the OS/2 kernel. A *keystroke monitor* is the fundamental part of a keyboard macro processor. A *printer monitor* is the fundamental part of a print spooler. A *mouse monitor* could be used in a custom pointer drawing routine.

16.1 What a Monitor Is

A device monitor is an ordinary process. The process registers as a monitor for a particular device. Then it reads incoming packets of data, and passes packets of data forward to their destination. For example, a keystroke monitor receives a packet for each keyboard make or break event. Only when it passes the packets forward will they reach the programs that are waiting to read from the keyboard.

There may be more than one monitor associated with a device. When there are, the monitors are organized into a chain, like a bucket brigade. The device driver stands at both ends of the chain; it hands the data to the first monitor, and takes the data delivered by the last monitor. In general, a monitor doesn't know whether the data it receives is direct from the device driver, or the output of another monitor. Nor does it know whether the data it passes forward will go to the device driver or will become input to another monitor.

A monitor must take pains to handle data as quickly as possible, so as not to slow down the system. Most monitors will create a separate thread that does nothing but receive and pass on data packets.

16.2 What a Monitor Does

At its simplest, a monitor merely looks at the data. It receives packets of data on its input side, examines them, and then passes them forward on its output side. For example, the first keystroke monitor in a chain receives each keyboard event as soon as the keyboard device driver prepares it. The monitor will usually examine the packet and pass it forward immediately. If there is another monitor, it will receive the packet as input, and will probably pass it forward.

When the last (or only) monitor in the chain passes a packet forward, the device driver takes it and sends it to its destination. In the case of a keystroke packet, the destination is the device driver's translation function and then the keyboard input buffer for the current screen group.

A monitor is not required to copy its input to its output. It is free to censor some packets by simply failing to pass them forward. It is free to alter the data it handles, passing forward modified data. And it is free to generate new data.

A keyboard macro processor, for example, would pass most keystrokes forward unchanged, but would replace certain ones with pre-recorded strings.

A print spooler, for example, would receive packets representing the output of every process that wants to print, mixed. It would sort them out, write some to disk, and send forward only the ones for the single file it was currently printing.

16.3 Structure of a Monitor

A monitor is an ordinary process that registers to be a monitor for a particular device. Once it has registered, it takes and passes forward data packets. When it is finished, it removes itself from the data stream.

16.3.1 Opening a Device For Monitoring

A monitor is set up in two steps. The first step is to get a handle associated with the device driver that is to be monitored. This is done with *DosMonOpen*, a simpler version of *DosOpen*. It takes the name of a character device. If that device driver supports monitors, a handle is returned that is associated with the device driver. If the driver does not support monitors (many do not), an error code is returned.

16.3.2 Registering the Buffers

Now the program must prepare two buffers, one for input and the other for output. Although these buffers are in storage owned by the program, their contents will be controlled by the device driver. The buffers represent the input and output sockets, so to speak, at which the monitor connects to the data stream.

The buffers are registered with a call to *DosMonReg*. This function takes the buffers, the handle from *DosMonOpen*, and two additional parameters. The first is a number that indicates where the process wants to be in the chain of monitors: the process may ask to be first in the chain, or last in the chain, or say it doesn't care.

The other word parameter to *DosMonReg* is referred to as an index but it is really a parameter to the device driver, and its meaning depends on the device driver. When dealing with the keyboard or mouse device driver, the index must be the number of the screen group whose keyboard or mouse is to be monitored. When dealing with the printer device driver, the index must be 1 for reasons that aren't entirely clear.

If *DosMonReg* returns zero, the buffers have been registered and the process is now a monitor. If it does not receive and pass forward data packets, all I/O to the monitored device will cease!

16.3.3 Ending the Monitor

When the process is finished monitoring, it can remove itself from the device stream with a call to *DosMonClose*. This function takes the handle used with *DosMonReg* and ends monitoring.

The call to *DosMonClose* is not strictly necessary; OS/2 should perform one automatically as part of closing all a process's open handles during termination. However, device monitoring is on the outer edge of the system where things tend to be a little strange. It is a *very* good idea to perform an explicit *DosMonClose* before terminating.

I have found that it does *not* work to call *DosMonClose* in an Exit routine scheduled by *DosExitList*. When control reached the exit code, the handle had already been closed and *DosMonClose* returned an error. It did work to put the *DosMonClose* in a handler for the termination signal as set up by

DosSetSigHandler. Then when the monitor program was stopping due to a control-break signal or a *DosKillProcess* call, the signal handler closed the monitor.

References: *DosExitList* 5.3.3; *DosSetSigHandler* 5.3.1

16.3.4 Monitor Input and Output

After it has registered, the main business of the monitor process is to receive packets and pass them on. It must do this without introducing any delays between input and output.

Ordinarily the best way to ensure speed is to use *DosCreateThread* to create a new thread of execution that will be devoted to monitor input and output. This is not required for a simple test of method, but in a production program it will probably be essential.

The monitor receives the next packet with a call to *DosMonRead*. This call is not like file input with *DosRead* (it doesn't take a handle, for one thing). It is a call on the device driver to say that the monitor is ready for the next packet. It takes as input the registered input buffer and a work buffer where the packet will be returned. Control returns when a packet is in the work buffer.

The form of a data packet depends on the kind of device being monitored. *DosMonRead* returns the size of the received packet in a separate parameter.

The monitor examines the received packet and decides what to do. Should it censor the packet? If so, it simply repeats the *DosMonRead*; the packet won't go any further in the chain.

Should it pass the packet forward, perhaps with modified contents? It does that with a call to *DosMonWrite*, naming the work buffer where the (modified) packet resides and the registered output buffer. This signals the device driver that a packet is being passed forward. If there's another monitor in the chain, and if it's waiting in a *DosMonRead* call, the packet will be moved directly to its work buffer. If the next monitor isn't waiting, the packet may be staged in the output buffer or in the input buffer of the next monitor (we aren't told). If there's no next monitor, the packet returns to the device driver to be processed.

The monitor might decide to expand the input packet. There's nothing to prevent it from doing many *DosMonWrite* calls, replacing one input packet with many

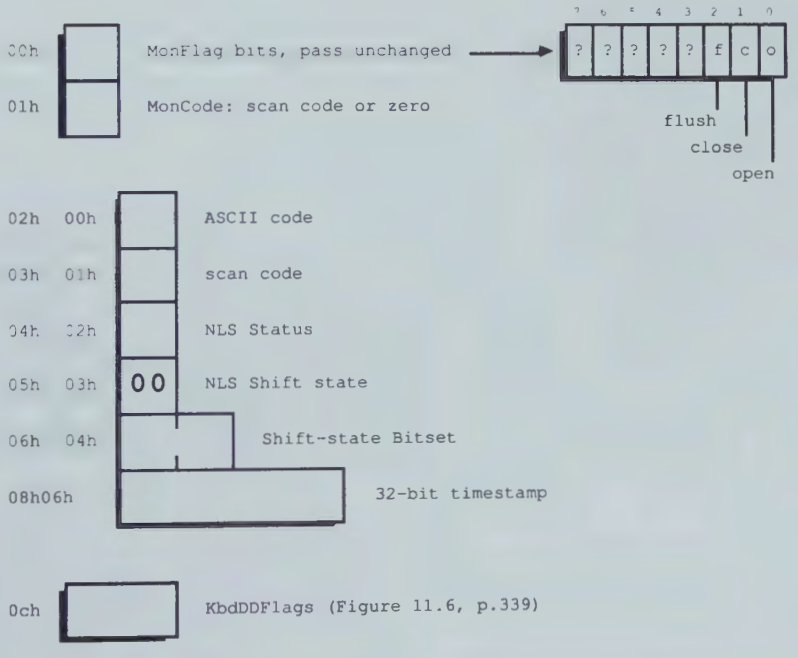
output packets.

Reference: creating threads 9.1.2

16.4 Keystroke Monitors

A keystroke monitor sees every keystroke make and break event generated by the keyboard hardware for a particular screen group. Each event is wrapped into a packet with the form diagrammed in Figure 16.1.

Figure 16.1
Keyboard Monitor Keystroke Packet



This structure has to be interpreted carefully. The first field must be examined first. If its least-significant bits specify a flush (from a *KbdFlushBuffer* call), open (from *KbdOpen*) or close (from *KbdClose*) event, then nothing in the rest of the packet is significant. These events may or may not be significant to the monitor, but they must be passed forward without fail; they must never be censored or modified.

monitor, but they must be passed forward without fail; they must never be censored or modified.

In other packets, the second byte usually contains a raw scan code as generated by the keyboard hardware. This is not identical to processed scan codes as returned by a *KbdCharIn* call, for example. Its most significant bit is zero for a make (key pressed down) and 1 for a break (key released). When a typamatic key is held down, it will generate a rapid series of make events, followed by a single break when it is released.

The last field, the device driver flags, is discussed in the topic on keyboard I/O. Its least-significant bits encode an important signal that reflects the device driver's interpretation of the packet. This signal number should be extracted and interpreted next (see Table 11.4, page 340). A keystroke monitor is registered for a particular screen group, and sees only the keystrokes that come to the logical keyboard of that screen group. This may seem to restrict the range of applications that can be written. If a monitor is doing something useful in one screen group, the operator will lose its services upon switching to another screen group!

This need not be the case. A keystroke monitor may register for any screen group, not necessarily the one in which the process is running. There is nothing to stop the process from registering a different pair of buffers for every possible screen group from 1 to 15 (screen group zero is the popup screen group and should not be monitored; use *DosGetInfoSeg* to find out the maximum screen group number). A global keystroke monitor would register buffers for each possible screen group, and then start one thread to read and write packets for each registered pair of buffers. Such a program could be started with a *run=* statement in the configuration file, and thus be available from the beginning.

Before you rush off to plan such a master program, consider a simpler solution. As you probably know, it is possible to make the command shell execute a batch command file whenever it starts up in a new screen group (in the IBM 1.0 version of OS/2 this command file is usually called *OS2INIT.CMD*). Write a keystroke monitor that monitors only the keyboard of its own screen group (use *DosGetInfoSeg* to get the number), but put a Detach command for that program in *OS2INIT.CMD*. A new instance of the monitor will be loaded and started whenever a new screen group is started. Little overhead will be involved since OS/2 doesn't reload the second and later instances of a loaded program.

A keystroke monitor receives a rapid flood of packets often in surprising sequences. Before trying to build an application that monitors the keyboard, you

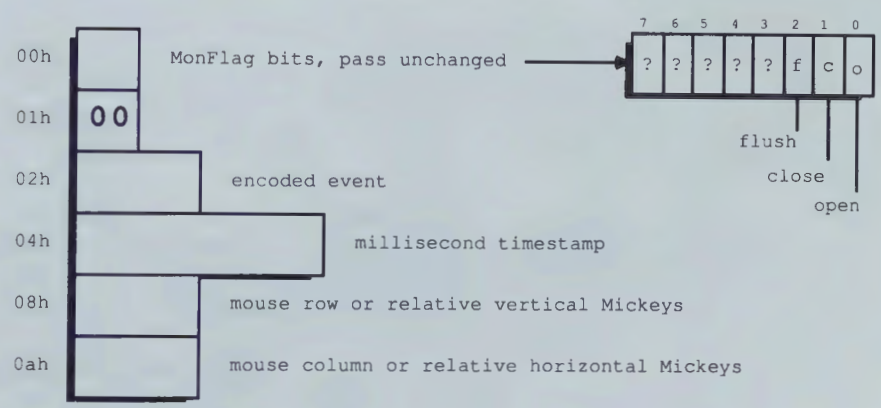
should write a test monitor and use it to explore the monitor data stream. An example keystroke monitor program can be found in the Disk Appendix to this book.

References: *KbdOpen* and *KbdClose* 11.3; *KbdFlushBuffer* 11.3.3; *KbdCharIn* 11.6; scan codes Table 11.3; device driver flag word Table 11.4; *DosGetInfoSeg* 5.1.4

16.5 Mouse Monitors

A mouse monitor, like a keyboard monitor, is registered to a particular screen group. Every mouse event that goes into the event queue for that screen group passes first through the monitor's buffers in the form shown in Figure 16.2.

Figure 16.2
Mouse Monitor Event Packet



The mouse monitor sees no activity until some process calls *MouOpen* in the screen group being monitored. Then it sees a packet containing only two bytes, with the *flush* bit set in the flag bits (not the *open* bit, for unknown reasons).

Once the mouse has been opened, the monitor sees all events in the form shown. The monitor sees motion data in the form that the process has requested using *MouSetDevStatus*. If the process requests motion data in relative Mickeys, that's what the monitor will see. If motion is reported by row and column, that is what is shown to the monitor as well. The monitor does not see any event that might tell it when the process has shifted from one mode of reporting to the other.

Tests with the example mouse monitor included in the Disk Appendix show that the monitor sees all mouse events as they are produced, not later when the process reads them with *MouReadEventQue*. It even sees events produced when the event queue is full, events that will be discarded.

It seems likely that a mouse monitor could be used as the basis of a custom pointer-drawing routine. The application would set up a high-priority thread to act as the mouse monitor and pointer-painter. Then it would open the mouse and use *MouSetDevStatus* to disable standard pointer drawing. Its monitor thread would paint the mouse pointer in the style and location desired.

References: *MovOpen* 12.2; *MovSetDevStatus* 12.3.2; *MovReadEventQue* 12.4.1; mouse pointer 12.1.3, 12.3.4

16.6 Print Monitors

A printer monitor sees a stream of packets that represent the mingled activities of all processes that are writing to that printer device. The printer device driver maintains two different streams of data for each device. One stream represents file activity: opens, closes, and *DosWrites*. This is the stream that is monitored when the index parameter to *DosMonReg* is given as 1.

The second stream, selected by an index of 2, is devoted to information related to switching code pages on the printer. This stream of data contains packets that represent commands to load new fonts into the printer, and to query the fonts that are there. It is primarily meant for the use of the Spool command.

If you use CodeView to trace the Spool command you can discover that it performs *DosMonReg* calls with index parameters of minus-1 (0ffffh). You can also find it passing a position-in-chain preference value of 3, a value which *DosMonReg* is not documented as supporting. The Spool command also receives access to the printer codepage information named in the devinfo statements of the configuration file. I have no more information on these matters to offer. If you want to monitor printer codepage switching, you have my best wishes.

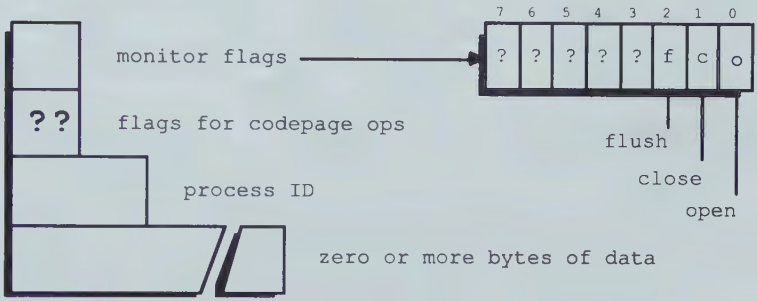
When monitoring printer stream 1, each packet contains the information diagrammed in Figure 16.3.

The first byte is a flag byte very similar to the one passed in a keystroke monitor packet, reflecting a flush, open, or close event.

The second byte is only applicable to stream 2, the codepage stream. If it is nonzero, the program is somehow monitoring that stream of data; it should pass the packet forward without change.

The third field of the packet contains the ID of the process that caused this activity. When multiple processes are writing to the same device, the packets they generate arrive in mixed order and the process ID field is the key to disentangling each substream.

Figure 16.3
Printer Monitor Packet



When one of the least significant three bits of the flag byte is nonzero, there will be no data after the process ID; the packet will be only four bytes long. When the low three bits of the flag byte are zero, the packet represents some amount of data being written to the monitored device. The size of the whole packet is returned by *DosMonRead*; the amount of data can be derived from that.

The data in a printer packet is not a neat unit like a byte or a line. It is simply a block of one or more (usually more) bytes on its way to the printer, with control characters and printable characters mixed. An example printer monitor is included in the Disk Appendix for this book.

Collected Error Codes

This is a quick reference to the error codes listed in the reference pages of this book. It is not an exhaustive collection of OS/2 error codes; it lists only errors that I have been able to evoke or that I am fairly sure will occur in common practice. If you discover more errors that ought to be listed, by all means notify the author at the address in the *Preface*.

Error Number	Returning Function	Probable Cause and Corrective Action:
1	DosFlagProcess	The second parameter must be zero or 1
1	DosHoldSignal	The action must be 0 or 1, and if 0 the count must be nonzero
1	DosSetSigHandler	The action must be in 0-4 and the signal number in 1-7
1	DosSetVec	The interrupt number must be 0, 4, 5, 6, 7 or 16
2	DosDelete	File not found (but path is OK; file may be a directory)
2	DosGetShrSeg	No segment of that name can be found
2	DosLoadModule	The file *modname*.DLL must exist in the *libpath*= directory
2	DosMove	File not found (but old path is OK)
2	DosOpen	File not found (but path is OK)
2	DosQFileMode	File not found (but path is OK)
2	DosSearchPath	No matching filename found on any path in the list
2	DosSetFileMode	File not found (but path is OK)
2	MouRegister	he dynamic link library couldn't be found
3	DosChDir	Path not found: invalid drive or a member doesn't exist or isn't a directory or the total path string exceeds 64 bytes
3	DosDelete	Path not found: invalid drive or a member doesn't exist or isn't a directory
3	DosFindFirst	Path not found: invalid drive or a member doesn't exist or isn't a directory

3	DosMkDir	Path not found: invalid drive or a member doesn't exist or isn't a directory
3	DosMove	One of the paths was no good (a member wasn't found or isn't a directory)
3	DosOpen	Path not found: invalid drive or a member doesn't exist or isn't a directory
3	DosQFileMode	Path not found: invalid drive or a member doesn't exist or isn't a directory
3	DosRmDir	Path not found: invalid drive or a member doesn't exist or isn't a directory
3	DosSetFileMode	Path not found: invalid drive or a member doesn't exist or isn't a directory
4	DosDupHandle	No available handles
4	DosLoadModule	Not enough file handles, use DosSetMaxFH to increase them
4	DosMakePipe	There are not two unused handles available
4	DosMonOpen	No available file handles (see DosSetMaxFH)
4	DosOpen	No available file handles
4	MouOpen	No handles are available, use DosSetMaxFH to create more
5	DosChDir	The last file named in the path isn't a directory
5	DosCreateCSAlias	The first parameter must select a private, read-write data segment
5	DosDelete	Access denied (file is read-only)
5	DosFreeSeg	That segment has already been freed or is a middle part of a huge allocation
5	DosGetSeg	The requested segment doesn't exist or wasn't allocated with the right option
5	DosGiveSeg	The specified segment doesn't exist or wasn't allocated with the proper option, or the specified process doesn't exist
5	DosLoadModule	*modname*.DLL exists but can't be opened for input
5	DosLockSeg	The specified segment doesn't exist or wasn't allocated with the proper option
5	DosMkDir	The last name in the path exists already
5	DosMove	The old file is a directory or the new file already exists
5	DosOpen	The specified file is a directory, or read-only, or the attribute parameter contains unsupported nonzero bits
5	DosReAllocHuge	The specified segment must be the first of a huge block
5	DosReAllocSeg	The specified segment doesn't exist
5	DosRmDir	The last name in the path isn't an empty directory
5	DosSetFileInfo	The file must be open for write access

5	DosSetFileMode	The target file is a directory
5	DosUnlockSeg	The specified segment doesn't exist or wasn't allocated with the proper option
6	DosBufReset	The handle must be open to a disk file
6	DosChgFilePtr	The handle must be open to a disk file
6	DosClose	The handle must be one returned by DosOpen and not yet closed
6	DosCloseSem	The handle must be one returned by DosOpenSem or DosCreateSem
6	DosDupHandle	One of the handles is not open
6	DosFileLocks	The handle must be open to a disk file
6	DosFindClose	The handle must be one returned by DosFindFirst and not closed
6	DosFindFirst	The input handle must be 1, -1, or an open handle returned by this call
6	DosFindNext	The handle must be one returned by DosFindFirst and not closed
6	DosFreeModule	The handle must that of a module bound to the current process
6	DosGetModName	The handle must specify a module now in storage
6	DosGetProcAddr	The handle must specify a module bound to the current process
6	DosMuxSemWait	All handles must be returned by DosOpenSem or DosCreateSem or valid far pointers
6	DosNewSize	The handle must be open to a disk file
6	DosQFHandState	The handle must be one returned by DosOpen and not yet closed
6	DosQFileInfo	The handle must be open to a disk file
6	DosQHandType	The handle must be one returned by DosOpen and not yet closed
6	DosRead	The handle must be one returned by DosOpen and not yet closed
6	DosSemClear	The handle must be one returned by DosOpenSem or DosCreateSem or a valid far pointer
6	DosSemRequest	The handle must be one returned by DosOpenSem or DosCreateSem or a valid far pointer
6	DosSemSet	The handle must be one returned by DosOpenSem or DosCreateSem or a valid far pointer
6	DosSemSetWait	The handle must be one returned by DosOpenSem or DosCreateSem or a valid far pointer

6	DosSemWait	The handle must be one returned by DosOpenSem or DosCreateSem or a valid far pointer
6	DosSetFHandState	The handle must be one returned by DosOpen and not yet closed
6	DosSetFileInfo	The handle must be open to a disk file
6	DosWrite	The handle must be one returned by DosOpen and not yet closed
6	Mou...	The handle is not an open one
8	DosAllocHuge	Not enough vacant storage or no more local selectors
8	DosAllocSeg	Not enough vacant storage or no more local selectors
8	DosAllocShrSeg	Not enough vacant storage or no local selectors
8	DosReAllocHuge	Not enough vacant storage
8	DosReAllocSeg	Not enough vacant storage
12	DosOpen	The access or share values in Mode word are improper
13	DosCWait	The first and second parameters must be 0 or 1
13	DosExitList	A parameter is wrong
13	DosKillProcess	The first parameter must be zero or 1
15	DosQCurDir	The drive number must exist in the system
15	DosQFSInfo	The drive number must exist in the system
15	DosSelectDisk	The drive number must exist in the system
15	DosSetFSInfo	The drive number must exist in the system
17	DosMove	The old and new paths must be on the same disk)
18	DosFindFirst	No files matched (count=0) or all matching files were returned
18	DosFindNext	No files matched (count=0) or all matching files were returned
19	DosWrite	The disk is write-protected
21	DosRead	Device not ready
21	DosWrite	Device not ready
22	DosDevIOCTL	The category and function codes must be ones the driver supports
22	DosRead	Device driver does not do reads (e.g. MOUSE$)
22	DosWrite	Device driver does not do writes (e.g. MOUSE$)
22	Mou...	The handle is not one opened by MouOpen
23	DosWrite	Read-check error
24	DosGetModName	The filespec string did not fit the buffer, try a larger buffer
24	DosSetSession	The structure length must be 6
24	DosStartSession	The first word of the structure must be its length, 24 decimal
25	DosRead	Error on seek to track
25	DosWrite	Error on seek to track

27	DosRead	Error on search to sector
27	DosWrite	Error on search to sector
29	DosWrite	Error on write
30	DosRead	Error on read
31	DosDevIOCTL	This "general failure" code is returned by the COM driver for almost any miscoding of parameter packet or data packet parameters
31	DosRead	General failure reported by device driver
31	DosWrite	General failure reported by device driver
32	DosDelete	Sharing violation (file is open in another process)
32	DosMove	The old file is open in another process and can't be moved
32	DosOpen	Another process has the file open with a conflicting access mode
33	DosFileLocks	The unlock range doesn't match a previous lock range, or the lock range overlaps on a locked area of the file
33	DosRead	The file is locked in the range that would be read
33	DosWrite	The file is locked in the range where the write would fall
34	DosRead	Disk volume has been changed since file was opened
34	DosWrite	Disk volume has been changed since file was opened
87	DosAllocHuge	The share option must be 0 or 1 and the greatest-size parameter must be at least as large as the allocation size
87	DosAllocSeg	The option word may not exceed 0007h
87	DosChgFilePtr	The method number must be 0, 1 or 2
87	DosCreateSem	The option must be 0 or 1
87	DosDevConfig	The inquiry number must be between 0 and 6 and the last parameter must be a word of zero.
87	DosFindFirst	The count must be nonzero and only the read-only, hidden, system, archive and directory attribute bits may be nonzero
87	DosFindNext	The count must be nonzero
87	DosMkDir	The last parameter must be a doubleword of 0
87	DosOpen	The Flag or Mode word is improper or the final doubleword is nonzero
87	DosReAllocHuge	The new size may not be larger than the maximum set with DosAllocHuge
87	DosRmDir	The second parameter must be a doubleword of 0
87	DosSetFHandState	Only the Write-through, Return-errors, and Inheritance bits may be nonzero
87	DosSetFileInfo	Only maintained fields may be nonzero

87	DosSetFileMode	Only supported attributes may be nonzero in the attribute word
87	DosSetMaxFH	The parameter must be no less than the existing number of handles and no more than 255
87	DosSetVerify	The parameter must be 0 or 1
87	MouGetPtrShape	The buffer size in the second parameter is too small (the minimum buffer size has been set in that word)
87	MouRemovePtr	The region coordinates may not exceed the size of the screen
87	MouSetDevStatus	Bits 15 through 10 must be zero
87	MouSetEventMask	Bits 15 through 7 must be zero
87	MouSetPtrPos	The coordinates must fall in the screen area
87	MouSetPtrShape	At least one of the fields of the second parameter is wrong
87	MouSetScaleFact	Both values must be nonzero
91	DosTimerAsync	System timer space exceeded
91	DosTimerStart	System timer space exceeded
100	DosCreateSem	The system limit on open semaphores has been exceeded
100	DosOpenSem	The system limit on open semaphores has been exceeded
101	DosSemClear	That's an exclusive semaphore and this process doesn't own it
101	DosSemSet	That's an exclusive semaphore and this process doesn't own it
101	DosSemSetWait	That's an exclusive semaphore and this process doesn't own it
102	DosCloseSem	This process owned the semaphore and didn't clear it
105	DosSemRequest	The previous owner terminated without clearing the semaphore
105	DosSemSetWait	The previous owner terminated without clearing the semaphore
105	DosSemWait	The previous owner terminated without clearing the semaphore
105	KbdSynch	Previous owner of the semaphore has terminated
106	DosOpenSem	The system limit on processes per semaphore has been exceeded
109	DosWrite	The handle specifies a pipe for which no read handles exist
110	DosMonOpen	The device named must exist and support monitors
110	DosOpen	Failure as requested by Flag word (file exists or doesn't)

111	DosFindFirst	The buffer size must be large enough for at least one entry
111	DosFindNext	The buffer size must be large enough for at least one entry
111	DosQCurDir	The buffer is too short for the result
111	DosQFSInfo	The buffer size must be at least 18 for function 1, at least 16 for function 2
111	DosQFileInfo	The buffer size parameter must be at least 22
112	DosNewSize	Insufficient disk space to resize file
112	DosOpen	Not enough disk space to create/resize file
121	DosMuxSemWait	The specified time elapsed and no semaphore cleared
121	DosSemRequest	The specified time elapsed and the semaphore was still not clear
121	DosSemSetWait	The specified time elapsed and the semaphore was still not clear
121	DosSemWait	The specified time elapsed and the semaphore was still not clear
121	KbdSynch	Semaphore not owned and wait option was 1 (zero wait timed out)
122	DosSetFSInfo	The length parameter is less than size of string
122	DosSetFileInfo	The size parameter must be at least 12
123	DosAllocShrSeg	The segment name must be an ASCIIZ string of the form "\SHAREMEM\segname.ext" and use characters valid in filenames
123	DosCreateSem	"\SEM\filename.ext" using only characters valid in a filename
123	DosGetShrSeg	The segment name must be an ASCIIZ string of the form "\SHAREMEM\segname.ext" and use only characters valid in filenames
123	DosLoadModule	The third parameter must be a filename of 1-8 characters
123	DosOpen	Filename or filetype is malformed
123	DosOpenSem	The name must be an ASCIIZ string of the form "\SEM\filename.ext" using only characters valid in a filename
123	DosSetFSInfo	The label string contains unpermitted characters
124	DosQFSInfo	The function number must be 1 or 2
124	DosQFileInfo	The second parameter must be 1
124	DosSetFSInfo	The function number must be 2
124	DosSetFileInfo	The second parameter must be 1
125	DosQFSInfo	That volume has no label (function 2)
127	DosGetProcAddr	The entry point must be one specified in the Linker EXPORTS list
127	MouRegister	The entry point can't be found in the library module

128	DosCWait	(last parameter zero) The current process has no remaining children
128	DosCWait	(last parameter nonzero) The designated process is not a child of the current process
129	DosCWait	(last parameter zero, second parameter 1) all children are active
131	DosChgFilePtr	With these parameters the file pointer would end up negative
132	DosChgFilePtr	Devices cannot have pointers changed
151	DosMuxSemWait	The event count must be between 1 and 16
152	DosMuxSemWait	The system limit on concurrent mux-waits has been exceeded
153	DosMuxSemWait	The list format is invalid (reserved word not zero?)
154	DosSetFSInfo	The label string may not have more than 11 bytes of text
155	DosCreateThread	The system can't create any more threads in this process
155	DosReadAsync	Not able to create a thread
155	DosWriteAsync	unable to create a thread
156	DosFlagProcess	The specified process has blocked these signals
156	DosKillProcess	That process has blocked the termination signal
157	DosLockSeg	Sorry, that segment was discarded
158	DosUnlockSeg	The segment wasn't locked (its reference count is zero)
162	DosFlagProcess	A signal of this type is already pending at that process
162	DosKillProcess	A termination signal is already being held for that process
183	DosAllocShrSeg	A segment of that name already exists
183	DosCreateSem	That semaphore already exists (open it instead)
186	DosFlagProcess	The third parameter must be zero, 1 or 2
187	DosOpenSem	No semaphore of that name can be found (create it instead)
203	DosOpen	Filename or filetype is malformed
203	DosScanEnv	Name not found in environment (remember: uppercase, no equals-sign)
203	DosSearchPath	Variable not found in environment
203	DosStartSession	The program file couldn't be found, specify a drive and pathname
206	DosOpen	Filename or filetype is too long
209	DosSetSigHandler	The signal number must be 1, 3, 4, 5, 6 or 7
303	DosFlagProcess	The first parameter must be an existing process
303	DosGetPrty	The process specified must exist
303	DosKillProcess	The second parameter must be an existing process

303	DosSetPrty	The process specified must exist
304	DosSetPrty	The priority change must lie between -31 and +31 inclusive
305	DosFlagProcess	When the option is zero, the process must be a child of this one
305	DosKillProcess	Option is zero but target process is not a child of this process
305	DosSetPrty	Only a process that is a child of the current process may be changed
307	DosSetPrty	The class must be 0, 1, 2 or 3
308	DosGetPrty	The first parameter must be zero or 2.
308	DosSetPrty	The option must be 0 1 or 2
309	DosGetPrty	A thread ID must identify an existing thread of the current process (if option=0, the first thread of the process has ended)
309	DosResumeThread	The thread must be part of this process, and suspended
309	DosSetPrty	The thread specified must exist in the current process
309	DosSuspendThread	The thread must be one that exists in this process
310	DosSubSet	With option 0 the size may not be less than the current size
311	DosSubAlloc	Not enough space remains in the segment
312	DosSubFree	The block described overlaps on other blocks
313	DosSubAlloc	The size parameter must be nonzero and less than the segment size
313	DosSubFree	The size word must be nonzero and smaller than the size of the segment
313	DosSubSet	The size parameter may not fall between 1 and 11 bytes
314	DosSubSet	The option must be 0 or 1
316	DosGetMessage	The message text had to be truncated to fit the buffer
316	DosInsMessage	The message text had to be truncated to fit the buffer
317	DosGetMessage	The message number must be defined in the message file
318	DosGetMessage	The message file could not be opened (might also be indicated by DosOpen error codes 2, 3, 4, 5, 32 or 123)
319	DosGetMessage	The message file named must be the output of MKMSGF
320	DosGetMessage	The size of the insertion table must lie between zero and 9

320	DosInsMessage	The size of the insertion table must lie between zero and 9
323	DosTimerAsync	Second parameter must be a handle returned by DosCreateSem or DosOpenSem
323	DosTimerStart	Second parameter must be a handle returned by DosCreateSem or DosOpenSem
324	DosTimerAsync	System timer space exceeded
324	DosTimerStart	System timer space exceeded
326	DosTimerStop	the handle must be one returned by DosTimerStart or DosTimerAsync
327	DosSetDateTime	At least one field is inconsistent or out of its proper range
330	DosPeekQueue	Only the queue's owner is permitted to do this
330	DosReadQueue	Only the queue's owner is permitted to do this
332	DosCreateQueue	A queue of that name already exists and may not be created
333	DosPeekQueue	There is no queue item following the number given in the fifth parameter (end of queue)
333	DosReadQueue	There is no queue item at the number given in the fifth parameter (number not from DosPeekQueue, or item already read)
335	DosCreateQueue	The queue name must start with "\QUEUES\" and continue with a properly formed filename and optional extension
335	DosOpenQueue	The queue name must begin "\QUEUES\" and end in a valid filename
336	DosCreateQueue	The queue-discipline number must be zero, 1 or 2
337	DosCloseQueue	The handle must be for an existing, opened queue
337	DosPeekQueue	The first parameter must be the handle of an existing queue
337	DosPurgeQueue	The handle must be of an existing queue created by this process
337	DosQueryQueue	The handle must specify an open, existing queue
337	DosReadQueue	The first parameter must be the handle of an existing queue
337	DosWriteQueue	The first parameter must specify an existing, open queue
342	DosPeekQueue	The queue is empty (and the wait option is 1)
342	DosReadQueue	The queue is empty (and the wait option is 1)
343	DosOpenQueue	No queue of that name exists
349	VioRegister	Invalid bit map (all zero?)
350	VioGetPhysBuf	The address or address plus length falls outside physical buffer
355	VioGetBuf	Display is in a graphics mode

355	VioSetMode	That combination of numbers isn't a supported display mode
355	VioShowBuf	The display is in a graphics mode
356	VioSetCurType	The cursor width must be 0 or 1
358	VioReadCellStr	The row number must be between 0 and screen height less 1
]358	VioReadCharStr	The row number must be between 0 and screen height less 1
358	VioScroll...	The bottom row number must be at least equal to top row number
358	VioSetCurPos	The row number must be less than screen height
358	VioWrt...	The row number must be between 0 and screen height less 1
359	VioReadCellStr	The column number must be between 0 and screen width less 1
359	VioReadCharStr	The column number must be between 0 and screen width less 1
359	VioScroll...	The right column number must be at least equal to left column number
359	VioSetCurPos	The column number must be less screen width
359	VioWrt...	The column number must be between 0 and screen width less 1
366	VioScrLock	The wait option must be 0 or 1
367	VioScrUnlock	The screen group isn't locked
369	DosSelectSession	The session number must be one returned by DosStartSession
369	DosSetSession	The session number must be one returned by DosStartSession
369	DosStopSession	The session number must be one returned by DosStartSession
370	DosStartSession	The system limit on screen groups has been exceeded
371	DosSelectSession	The session number must be one that is still active
371	DosSetSession	The session number must be one that is still active
371	DosStopSession	The session number must be one that is still active
375	KbdCharIn	The wait option must be 0 or 1
376	KbdGetStatus	The structure length field must contain the structure length
376	KbdSetStatus	The shift state or another field of the structure is invalid
376	KbdStringIn	The buffer-size word may not exceed 255
377	KbdSetStatus	The Echo bits may not both be set
378	KbdSetStatus	The Raw and Cooked bits may not both be set

379	DosMonRead	The wait option is not 0 or 1, or the monitor buffer is not a registered buffer, or monitoring has been ended with DosMonClose
379	DosMonReg	Something is wrong with one of the parameters
379	DosMonWrite	The first parameter isn't a registered output buffer or monitoring has been ended by DosMonClose
381	DosMonClose	The handle must have been returned by DosMonOpen and still be open
381	DosMonReg	The handle wasn't opened with DosMonOpen
382	DosMonRead	The size given is less than the packet size
383	DosMonRead	No packet is available and the wait option was 1
384	DosMonWrite	The packet size must be less than or equal to the size of the registered buffer
385	MouOpen	The mouse device driver was not started at configuration time
395	DosBeep	The pitch value must be between 32 and 32,767
396	DosCaseMap	Couldn't locate the COUNTRY.SYS file
396	DosGetCollate	Couldn't locate the COUNTRY.SYS file
396	DosGetCtryInfo	Couldn't locate the COUNTRY.SYS file
397	DosCaseMap	Couldn't open the COUNTRY.SYS file
397	DosGetCollate	Couldn't open the COUNTRY.SYS file
397	DosGetCtryInfo	Couldn't open the COUNTRY.SYS file
398	DosCaseMap	The country number must be a supported one
398	DosGetCollate	The country number must be a supported one
398	DosGetCtryInfo	The country number must be a supported one
398	DosSetCP	The code page must be a supported one
399	DosGetCP	The data that to be returned was longer than the buffer and was truncated
399	DosGetCollate	The buffer wasn't large enough for the entire data structure
399	DosGetCtryInfo	The buffer wasn't large enough for the entire data structure
399	DosGetDBCSEv	There was more data to return than would fit
402	VioPrtSc	Only the session manager may do this
404	VioDeregister	No display subsystem was registered by this process
405	VioEndPopup	No pop-up active
406	VioPopup	The pop-up screen group is in use (and first parameter is zero)
409	KbdRegister	Bits 14 and 15 of the bit map must be zero
410	KbdRegister	A registered subsystem is already active in this screen group
411	KbdDeregister	No keyboard subsystem has been registered by this process

414	MouRegister	Bits 22 through 31 of the doubleword mask must be zero, and the mask must be nonzero
415	MouRegister	Only one subsystem may be registered per screen group
416	MouDeregister	This process hasn't registered a mouse subsystem
421	VioGetConfig	The first parameter must be zero
421	VioGetState	The request type must be 0, 1 or 2; the first register number must lie between zero and 15
421	VioSetANSI	The set value must be 0 or 1
421	VioSetState	Request type must be 0, 1 or 2; first palette register and border color must be in 0-15; switch value must be 0 or 1
422	VioModeWait	Only one process per screen group may do this
422	VioSavRedrawWait	Only one process in a screen group may do this
423	VioModeWait	Wait ended by a VioModeUndo call
423	VioSavRedrawWait	Wait ended by a VioSavRedrawUndo call
426	VioRegister	Only one VioRegister allowed per screen group
427	VioModeUndo	No ModeWait thread has been established by this process
428	VioSavRedrawUndo	No Redraw watchdog has been established by this process
429	VioGetPhysBuf	This may only be done when the screen group is in the foreground
430	VioGetBuf	This may not be done during a pop-up display
430	VioGetPhysBuf	This may not be done during a pop-up display
430	VioPopup	This may not be done during a pop-up display
430	VioShowBuf	This may not be done during a pop-up
435	MouReadEventQue	The second parameter must be zero or 1
436	VioGetPhysBuf	The second parameter must be zero
436	VioSetCP	The display handle and first parameter must be 0
436	Vio...	The display handle must be zero
438	VioGetConfig	The structure length field must be 4, 6 or 10
438	VioGetFont	Structure length must be 14 and the font space must be large enough for the entire font (256 times number of rows in a box).
438	VioGetState	Structure length must agree with request type and with implied number of registers for request type 0
438	VioSetFont	Structure length must be 14; font space must be large enough for the entire font; pel rows and columns must be reasonable
438	VioSetState	The structure length word must match request type (and implied number of registers in request type 0)

439	Kbd...	The keyboard handle must be 0 or one returned by KbdOpen
440	KbdOpen	No more handles available
442	KbdSetCP	The codepage couldn't be read from disk
443	KbdSetCP	The codepage number is invalid
444	KbdSetCP	Code page support was not set up during configuration
445	Kbd...	The keyboard focus is required
446	KbdGetFocus	The focus is on another keyboard (and wait option was 1)
448	KbdSetCP	Selected code page has an invalid format
448	KbdSetCustXT	Invalid code page format
453	DosStartSession	The "foreground" option must be zero or 1
454	DosStartSession	The "related" option must be zero or 1
455	DosSetSession	The "Binding" option must be zero, 1 or 2
456	DosSetSession	The "Selectable" option must be zero, 1 or 2
464	Kbd...	A detached process has no keyboard access
465	Vio...	A detached process has no display access
466	MouOpen	A detached process has no mouse available
467	VioGetFont	No font available of the requested dimensions
467	VioSetFont	Second word must be zero; character dimensions must be supported by the attached hardware
467	VioSetMode	There's no font available to support that mode
468	VioGetCP	Program-defined font is active
469	VioSetCP	The codepage number must be zero or defined in "country" statement
470	VioSetCP	The hardware doesn't support loadable fonts
478	DosStartSession	The "trace" option must be zero or 1

Index to System Calls

This special index shows where each OS/2 system call is discussed in Chapters 1-16. For concise reference information on a system call, look it up by name in the reference section. For references to more general terms and phrases, see the regular *Index* that begins on page 453.

DosAllocHuge, 201, 207
DosAllocSeg, 93, 201, 203, 206-7
DosAllocShrSeg, 206, 282
DosBufReset, 235, 242-43
DosCaseMap, 161
DosChDir, 217-19
DosChgFilePtr, 225, 238, 240-42
  with lock, 247
DosClose, 235, 237
  forces output, 242
  serial port, 393-95
DosCloseSem, 267
DosCreateCSAlias, 59
DosCreateQueue, 192, 280-81
DosCreateSem, 267
DosCreateThread, 130, 250-51, 254-58, 428
DosCSAlias, 196
DosCWait, 182-83, 193
DosDelete, 220, 226
DosDevConfig, 139
DosDevIOCTL, 33-34, 324, 375-77, 421-23
  calling in C, 401
  calling in Pascal, 403
  category 1 function summary, 402
DosDupHandle, 187-89, 215, 235, 237-38
DosEnterCritSec, 169, 260
DosExecPgm, 182-85, 191, 325, 403
DosExit, 130, 165, 251
DosExitCritSec, 260
DosExitList, 173-74, 275, 278, 427-28
  restore display, 314
DosFileLocks, 245-47
DosFindFirst, 211, 221, 223-28
  handling multiple returns, 226-27
  with other file calls, 225-26
DosFindNext, 223-27
DosFlagProcess, 163, 167, 193, 283
DosFreeSeg, 205
DosGetCollate, 161
DosGetCP, 161
DosGetCtryInfo, 158-60
DosGetDateTime, 142, 146
DosGetDBCSEv, 162, 328
DosGetDevConfig, 141-42
DosGetEnv, 129, 154, 157
DosGetHugeShift, 202
DosGetInfoSeg, 140, 145-54, 202, 268, 430
DosGetMachineMode, 139-40
DosGetMessage, 177-79
DosGetModHandle, 133-35
DosGetModName, 129, 134
DosGetPID, 145, 153
DosGetProcAdr, 134-37
DosGetPrty, 252
DosGetSeg, 206-7, 282
DosGetShrSeg, 206
DosGetVersion, 139-40, 146
DosGiveSeg, 193, 207
  display buffer, 310
DosHoldSignal, 164
DosInsMessage, 177-78
DosKillProcess, 130, 163, 165-66, 170, 193
DosLoadModule, 131-35
DosLockSeg, 204

DosMakePipe, 188-90
DosMemAvail, 201
DosMkDir, 211, 219
DosMonClose, 427-28
DosMonOpen, 426-27
DosMonRead, 428, 433
DosMonReg, 427, 432
DosMonWrite, 428
DosMove, 220, 226, 235
DosMuxSemWait, 266
DosNewSize, 225, 236
DosOpen, 215-16, 225, 228-35
  action word, 230
  and device driver, 373
  change file size, 236, 243
  device names, 228-29
  disk volume, 420
  flag word, 230
  inheritance flag, 186-87
  mode word, 231-33
  screen output, 290
  serial port, 393-95
  write-through, 242
DosOpenQueue, 193, 281-82
DosOpenSem, 267
DosPeekQueue, 283
DosPTrace, 184
DosPurgeQueue, 284
DosPutMessage, 178
DosQCurDir, 217
DosQCurDisk, 141, 217
DosQFHandState, 236
DosQFileInfo, 227-28, 234
DosQFileMode, 221, 227
DosQFSInfo, 141, 218-19, 223, 225, 244
DosQHandType, 236-37, 289, 325
DosQueryQueue, 284
DosQVerify, 215
DosRead, 238-41
  performance, 243-45
  serial port, 398-401, 411
  with lock, 247
  with pipe, 188-90
DosReadAsync, 241, 249
DosReadQueue, 283-84
DosReallocHuge, 203
DosReallocSeg, 88, 93, 203
  expanding environment segment, 157
DosResumeThread, 251
DosRmDir, 211, 219-20
DosScanEnv, 155-56, 403
DosSearchPath, 156, 221-22
DosSelectDisk, 217
DosSelectSession, 194
DosSemClear, 262, 265-66, 272-73
DosSemRequest, 262, 267, 272-73, 276,
  in replaced display function, 323
DosSemSet, 265, 273
DosSemWait, 265-66, 272
DosSetCP, 161
DosSetDateTime, 142
DosSetFG, 196
DosSetFHandState, 187, 189, 233, 236
DosSetFileInfo, 223, 227-28
DosSetFileMode, 213, 221, 223, 226-27
DosSetFSInfo, 219, 223
DosSetMaxFH, 214-15
DosSetPrty, 252-53
DosSetSession, 195
DosSetSigHandler, 165-68, 428
DosSetVec, 169-73
DosSetVerify, 215
DosSGSwitch, 196
DosSGSwitchMe, 196
DosSGSwitchProc, 196
DosSleep, 264, 269
  pace scrolling, 309
  timing break, 415
  with file locks, 246
DosStartSession, 191-94
DosSubAlloc, 203
DosSubFree, 203
DosSubSet, 203
DosSuspendThread, 251
DosSystemService, 185, 196
DosTimerAsync, 147, 269
DosTimerStart, 147, 269
DosTimerStop, 269
DosUnlockSeg, 204
DosWrite, 240-43
  during pop-up, 311
  performance, 243-45
  printer stream, 432
  serial port, 396-98
  to display, 289-91
  vs. DosPutMessage, 178
  with lock, 247
  with pipe, 188-90
DosWriteAsync, 241, 249
DosWriteQueue, 193, 282
KbdCharIn, 333-37, 430
KbdClose, 329-30, 429

KbdCustCP, 337
KbdDeregister, 341-43
KbdFlush, 333
KbdFlushBuffer, 330-31, 429
KbdFreeFocus, 329-30
KbdGetCP, 337
KbdGetFocus, 329-30
KbdGetStatus, 331-33
KbdOpen, 329-30, 333, 429
KbdPeek, 326, 333-37
KbdRegister, 341-43
KbdSetCP, 161, 337
KbdSetStatus, 331-33
KbdStringIn, 333
KbdSynch, 343
KbdXlate, 338-40
MouClose, 367
MouDeregister, 367-70
MouDrawPtr, 362-63
MouFlushQue, 366
MouGetDevStatus, 357-58
MouGetEventMask, 364
MouGetHotKey, 357
MouGetNumButtons, 357
MouGetNumMickeys, 357
MouGetNumQueEl, 365
MouGetPtrPos, 366-67
MouGetPtrShape, 359-62
MouGetScaleFact, 358-59
MouOpen, 354-56, 362, 364, 366, 431
MouReadEventQue, 363-66, 432
MouRegister, 367-70
MouRemovePtr, 362-63
MouSetDevStatus, 358, 365, 431
MouSetEventMask, 364
MouSetHotKey
  (not documented)
MouSetPtrPos, 367
MouSetPtrShape, 359-62
MouSynch, 369
VioDeRegister, 321
VioEndPopUp, 311
VioGetANSI, 288-89, 291, 293-94
VioGetBuf, 310
VioGetConfig, 294-95
VioGetCP, 294, 301
VioGetCurPos, 293-94, 299, 313
VioGetCurType, 293-94, 299-300, 313
VioGetFont, 294, 300-2
VioGetMode, 142, 293-94, 296-97, 300, 313
VioGetPhysBuf, 314-15
VioGetState, 294, 303, 313
VioModeUndo, 320
VioModeWait, 320
VioPopUp, 311
  replacing, 321
VioReadCellStr, 307, 313
VioReadCharStr, 307
VioRegister, 321-23
VioSavRedrawUndo, 320
VioSavRedrawWait, 319-20
VioScrLock, 316, 319
VioScrollDn, 308
VioScrollLf, 308
VioScrollRt, 308
VioScrollUp, 308
VioScrUnlock, 316
VioSetANSI, 291, 293-94
VioSetCp, 161, 294, 301
VioSetCurPos, 294, 299, 313
VioSetCurType, 293-94, 300-1, 313
VioSetFont, 294, 302
VioSetMode, 293-94, 296, 298, 301, 313
  resets cursor, 301
  resets font, 303
VioSetState, 294, 303, 313
VioShowBuf, 310
VioWrtCellStr, 307, 313
VioWrtCharStr, 307-8
VioWrtCharStrAtt, 307
VioWrtNAttr, 307
VioWrtNCell, 307
VioWrtNChar, 306-7
VioWrtTTY, 290-92, 294, 311

General Index

This is a general index to the terms used in Chapters 1-16. The OS/2 system calls are indexed separately in the *Index to System Calls* beginning on page 449. The system calls are also listed alphabetically in the reference section.

80286, 17, 37. *See also* CPU
80287. *See* Numeric Coprocessor
80386, 17, 37. *See also* CPU

accent keys, 327-28
access rights byte.
  See segment descriptor
address, 58
address space, 21, 54-70
  belongs to process, 19
  Protect mode, 21, 55-70
  Real mode, 21, 54-55
ADS, 136, 146
ADSPROC, 136
allocation. *See* storage allocation.
allocation unit
American National Standards Institute (ANSI), 380
ANSI escape sequences, 287, 291-93
  table of, 292
ANSI.SYS, 374
application, 17
array bounds trap, 171
ASCII
  case mapping, 161
  collating sequence, 161
  double-byte characters, 162
  ENQ and ACK, 390
  keyboard data, 334
  keyboard translation, 327-28
  SUB, 400-1
  XON and XOFF, 390-91
ASCIIZ, 154, 212

assembly language
  common segments in, 79
  communal names in, 80
  DGROUP, 91-94
  external names in, 78
  linking names in mixed case, 102
  near and far labels, 48
  program entry point, 81
  public names in, 77
asynchronous I/O, 24, 241
asynchronous processing, 28, 249-80
  case study, 269-80
  critical sections in, 260, 263
  overview, 258-59
  polling, 263-65
  queues, 280-84
  semaphore as key, 262-63
  semaphore as signal, 265-66
  semaphore fundamentals, 261
  system semaphore, 30, 266-67
  timer services, 268-69
attribute. *See* display, file attributes

background, 18
binary image
  in executable file, 119, 124
  in object module, 76
binding, 76-80
  between modules, 77-80
  between segments, 76-77
BIOS
  display modes, 296
  display output, 285

level of, 141
BIOS Parameter Block (BPB). *See* disk I/O
bit, 38
BOUND instruction, 171
byte, 38

C language
command parameters, 157, 185
communal names, 79-80
conventional segments, 89-91
creating threads, 255-58
declaring
directory scan records, 227
disk info record (BPB), 422-423
display state parameter, 303
OS/2 functions, 48
semaphores, 262
signal handler, 166
thread stack, 255
DGROUP with, 91-94, 255
DosDevIOCTL, 401
external names in, 78
heap use, 93
indirect calls, 135-36
letter case of names, 102-3
near and far functions, 48
opening screen as file, 290
OS/2 version stored by, 141
program entry point, 81, 129
public names in, 77
small model, 48
stack allocation, 47
cache
call gate, 68-69
calling convention. *See* parameter convention
calling indirect addresses, 134-37
case conversion, 161
character cell, 286, 304-6
attribute values, 305-6
size of, 300, 302
child process, 26, 182-90
creation of, 182-84
detached, 184
environment segment of, 185-86
file handles inherited, 186-87
pipes created for, 189-90
standard files, 186-88
child session, 191-95
class of segment, 84
clearing screen, 308
client program, 73
CMD.EXE, 194
code page. *See* National Language Support
code segment, 29
alias of data, 59, 196
alignment, 83
combined by link, 85-86
conventional, 89-91
definition file, 110
execute-only, 113
I/O privileged, 113-17
collating sequence, 161
Color Graphics Adapter (CGA), 287, 295
COM device. *See* serial port
COM filetype, 51
combine type, 84-85
command parameters, 157, 185
command shell, 182, 194
common segment, 78-79, 84
communal name, 79, 92
compatibility mode. *See* MS-DOS
CON device name, 229
configuration data at run time, 150-51
configuration file
codepage, 328, 337
country, 158
device, 372, 391
devinfo, 328
libpath, 132
maxwait, 254
memman, 58, 197-98
priority, 151, 254
protectonly, 151
protshell, 182, 195
rmsize, 197
run, 153
timeslice, 151, 253-54
conforming segment, 67, 113-14
control key signals
-C and -Break, 130, 162-68, 274
-P and -PrtSc, 291
-Z, 238, 240, 400-1
in keyboard DDflags, 340
with keyboard modes, 332
country numbers, 160
country-dependent data, 158-62
COUNTRY.SYS file, 158
figure, 159
CPU, 63. *See also* trap
addressing, 54-70
flag word, 67

flag word, 67
invalid opcode, 171
machine status, 140, 172
Protect mode, 21, 55-70
Real mode, 21, 54-55
registers, 39-50
critical section, 260
currency format, 158-60
current directory, 211-12
belongs to process, 19, 212
in path search, 222
longest string, 218
query and set, 217-18
current drive, 210-11
belongs to process, 19, 211
in path search, 222
query and set, 217
Current Privilege Level (CPL), 60, 66-67

DASDOpen. *See* disk volume I/O
data segment, 29
alias as code, 59, 196
alignment, 83
combined by link, 86
conventional, 89-91
definition file, 110-13
read-only, 112-13
date and time, 30, 142-43, 146
from CLOCK$, 239
date format, 158-60
date of file, 212
debug data in program, 94-95
debug process created, 184
default directory. *See* current directory
default drive. *See* current drive
defined segment, 75
definition file, 73, 99, 107-17
CODE, 110, 122
CONFORMING, 113-14
DATA, 110-11, 122
DESCRIPTION, 107, 123
EXECUTEONLY, 113
EXECUTEREAD, 113
EXPORTS, 114-15, 123, 134
HEAPSIZE, 110, 122
IMPORTS, 114-15
IOPL, 113-14
LIBRARY, 109, 123, 126, 132-33
LOADONCALL, 112
NAME, 109, 123
PRELOAD, 112
PROTMODE, 109-10
READONLY, 112-13
READWRITE, 112-13
REALMODE, 109-10
RESIDENTNAME, 116, 123
SEGMENTS, 111-12, 122
STACKSIZE, 110
STUB, 109-10
summary, 107-9
deleting files, 220
descriptor
detached process
CON equal to NUL, 229
created, 184
methods of detecting
KBD$ is null device, 325-26
KbdPeek fails, 326
MouOpen fails, 355
parent PID is zero, 145
VioGetANSI fails, 288
no display access, 288
no keyboard access, 326
no mouse access, 355
screen group number, 153
device driver, 33-34, 371. *See also* serial
disk volume
and DosClose, 235
and DosOpen, 373
attribute word, 236-37
block vs. character, 372-73
fake, 374
keyboard, 326
loading, 371-72
mouse, 352
mouse pointer, 353, 355
operation, 373-74
device I/O
CLOCK$, 229, 239
CON, 239-40
KBD$, 229, 325-50
MOUSE$, 229, 240-41
printer, 229, 240
SCREEN$, 229, 240, 290
device monitor, 34, 425-33. *See also*
keyboard, mouse, printer monitor
input and output, 428-29
opening, 426-27
operation of, 425-26
registering, 426-27
structure, 426-27
device names

device names
to DosOpen, 228-29
DGROUP, 91-94
at entry to process, 129
dynamic link startup, 132
figure, 92
formed by Link, 87
heap size in, 93, 110
in definition file, 111
in executable file, 122
in exit list, 174
loading address, 62, 76
segment order, 104
stack size in, 93, 105-6, 110
directory, 211
creating, 211, 219
current, 211-12
dates and times in, 223-24
deleting, 211, 219
dot and doubledot, 211
entry format, 224
root, 211
scanning, 223-27
discarding segments, 58, 127-28, 198-99
disk
allocation, 240, 242-43
allocation unit size, 218
arm contention, 244
cache, 242, 244-45
capacity, 218
information, 218-19
opened as file, 24, 419-24
organized like MS-DOS, 23
query removable, 424
sector size, 218, 422-23
swap disk, 200
disk label
not in directory scan, 213, 223
query and set, 219
disk volume, 209, 218-19
disk volume I/O, 419-24
locking, 420-21
opening, 420
querying BPB, 218-19, 421-24
querying removable, 424
summary table, 419
diskette drive
number of, 141
dispatch class, 28, 252
in local info segment, 153
query and set, 252-53
dispatching. *See* scheduling
display, 31, 285-324
adapter types, 287, 295
border color, 304
character cell, 286
clear, 308
code page, 301
font, 301
in screen group, 18, 287-88
logical buffer, 309-10
modes, 286-87, 296-98
pages, 288
pallette registers, 303
physical buffer, 23, 285, 314-16
replacing functions, 321-24
save and restore, 313
scrolling, 308-9
display attribute, 286, 305-6
bit-7 switch, 304
byte format, 305-6
display configuration, 295
display cursor, 299-301
hiding, 300
size and shape, 300
display mode, 296-98
color burst bit, 297
color, not monochrome bit, 297
graphic mode bit, 297
graphic mode table, 299
hardware addresses, 315
number of colors, 297
text mode table, 298
display output -- graphics, 313-20
in logical buffer, 310
in physical buffer, 314-16
operation, 285
programming registers, 317-18
watchdog threads, 318-20
display output -- text
ANSI sequences, 291-93
functions, 306-7
hardware operation, 286-87
logical buffer, 309-10
opening as file, 290
scrolling, 308-9
stream output, 290-91
using standard handles, 289
display state functions, 294-304
summary, 294
disposable segment, 203-4
divide by zero trap, 170

DOS box. *See* MS-DOS
DOS segment order
Double-Byte Character Sets (DBCS), 162,
doubleword
duplicating handles, 187, 189
dynamic link, 25, 73
  definition file, 116
  in executable file, 124
  in object file, 80
  load time, 124-25
  ordinal numbers, 108, 116, 134-35
  run time, 133-37
dynamic link library (.DLL), 25
  definition file, 109, 111, 115-17
  format same as .EXE, 119
  initialization, 126, 131-33
  linking, 98
  loading, 125-26, 132-33
dynamic storage allocation. *See* storage allocation

EGA.SYS, 374
Enhanced Graphics Adapter (EGA), 287, 295
  pallette registers, 303, 317-18
entry point
environment segment, 26
  address at program entry, 129
  address at run time, 154
  belongs to process, 19
  command parameters in, 157
  created by DosExecPgm, 185-86
  format of, 154-58
  updating or expanding, 157-58
environment variables, 23, 26
  COMSPEC, 194, 403
  DPATH, 221
  format of, 155
  LIB, 99
  locating at run time, 155-56
  PATH, 221
  path lists, 156, 221-22
essential segment, 74
exception. *See* trap
EXE file. *See* executable file
executable file, 73, 119-25
  alignment of segments in, 104
  debug data in, 94-95
  dynamic link and program same, 119
  held open while loaded, 127-28
  loaded module takes priority, 127
  module-table data, 121-25
  MS-DOS form, 25, 109, 119-21
  OS/2 form, 24-25, 109, 120-25
  output of Link, 98
  stack size, 105, 110
exit list, 27, 173-74, 275, 314
exported name, 77, 114-15
  address at run time, 134-137
extended keyboard input, 334
external reference, 73, 78

family programs, 24, 110
far call and jump, 47-48, 63
  optimized by link, 85
  privilege rules, 67-68
  via call gate, 68, 115
far pointer, 46
file
  closing, 235
  deleting, 220
  input, 238-40
  locking, 24, 245-47
  moving, 219-20
  opening, 228-35
  output, 240-43
  renaming, 219-20
  sharing, 18, 21, 232
file attributes, 213-14
  query and set, 221
  set at open, 230
file date and time
  creation, last-access not kept, 223-24
  format of, 212-13
  query and set, 227-28
  returned by directory scan, 223-24
file handle. *See* handle
file I/O
  asynchronous, 24, 241
  input, 238-40
  output, 240-42
  performance, 216, 243-45
  read-write pointer, 238
  verify switch, 216
file name format, 210
file open, 228-35
  action returned, 230
  attribute, 234
  common scenarios, 234-35
  device name, 228-29
  file size, 234
  flag word, 229-30
  mode word, 231-33

mode word, 231-33
access mode, 231
inherit bit, 232
return errors, 233
share option, 232
write through, 233
pathname, 228
file sizes, 224-25
changing, 235-36
set at open, 230
file system ID, 218-19
fix-up records, 120, 124-125
fixed disk, 17
flow control. *See* serial port
foreground, 18, 151
FORTRAN language, 79
function keys, 325, 335, 344-46

generic IOCTL. *See* I/O Control
Global Descriptor Table (GDT), 22, 60
selector bit 2, 61-62
global information segment, 145-52
configuration data, 150-51
date and time, 146
figure, 148
foreground screen group, 151
huge shift, 146
number of screen groups, 151
OS/2 version, 146
process ID using keyboard, 152
timer resolution, 147
timestamps, 147-50
trace flags, 152
graphics mode display, 31-32
Greenwich Mean Time (GMT), 142-44
group, 87. *See also* Link program

handle
duplication of, 187, 189, 215, 237-38
in directory scan, 223
in IOCTL, 375
inherited by child, 182, 186-88
maximum number of, 215
mode bits, 231-33
query and set, 236
of file, 214-15
of keyboard, 329
of module, 129, 132-34
of mouse, 354-55
of semaphore, 267
table of, per process, 19, 214-15
testing file/device/pipe, 236-37
handler
hardware trap, 169-73
software signal, 166-68
hard disk. *See* fixed disk
hardware trap. *See* trap
heap, 93, 110
hexadecimal, 38
hot key, 19, 191, 288
huge model, 53-54, 90
huge segment, 53-54, 90, 201-2, 207
huge shift, 146, 202

I/O Control (IOCTL), 33-34, 374-77. *See also*
disk volume I/O; serial port I/O
categories, 376
data packet, 377
generic, 375
handle, 375
parameter packet, 377
I/O Privileged segments
definition file, 69-70, 113-14
loading of, 124-25
programming display registers, 318
reached via call gates, 113-14
imported name, 78, 114-15
alias in IMPORTS, 115
ordinal vs. name, 116, 134-35
inheritance, 182, 186-88
input. *See* file, device, and serial port I/O
interrupt. *See* trap
INTO instruction, 170-71

keyboard, 17, 32, 325-50
accent keys, 327-28
buffer, 330-31
code page, 327-28
functions replaced, 341-43
id of process using, 152
in screen group, 18
scan code tables, 344-50
scan codes, 326
sharing, 328-30
shift states, 327, 332, 336-37
string input, 333
SysReq key, 337
template editing, 333
test if available, 325-26
translation, 338-40
keyboard controller, 22

keyboard data record, 334-37
  ASCII byte, 334
keyboard focus, 32, 329-30
keyboard monitor
  data packet, 429-31
  design considerations, 430
  device driver flags, 339-40
keyboard status, 331-33
  cooked mode, 332
  echo mode, 331
  raw mode, 332
  turnaround character, 332

large model, 90, 93
LIB environment variable, 99
link map, 98, 101-2, 104
Link program, 26, 95-117
  combines segments, 82-89
  forms groups, 87
  input, 73-81
  option summary, 101
  option syntax, 100
  options, 100-6
    ALignment, 104, 124
    CODeview, 104
    DOSSeg, 104
    FARCalltranslation, 85-86, 105
    HElp, 100
    INformation, 100
    LIne, 104
    MAP, 98, 101-2
    NODefaultlib, 103
    NOIgnorecase, 102-3
    NONull, 104
    NOPack, 88, 105
    PACkcode, 105
    STack, 105, 122
  packs segments, 87-88, 105
  parameter summary, 97
  parameters, 95-99
    definition file, 99
    entry of, 96-97
    libraries, 99
    map file, 98
    object filenames, 98
    response file, 96-97
  resolves bindings, 82
linking, 71-95
load-time dynamic linking, 73
loaded module or program. *See also* module table
  files held open, 127
  handle in DI at entry, 129
  multiple instances, 127
  priority over files, 127
  reference count, 126
loading, 71-72, 125-26. *See also* module
  executable format, 119-25
  fixing up references, 124-25
  IOPL entries, 124-25
  preload vs. load-on-call, 112
  reload after swap, 198-99
Local Descriptor Table (LDT), 22, 25, 61
  assignment of slots in , 64-66
  dynamic storage allocation, 202
  selector bit 2, 61-62
local information segment, 145, 152-54
  figure, 153
  process ID, 153
  scheduling priority and class, 153
  screen group number, 153
  thread ID, 153
lock range, 245

machine status word
  testing NPX present, 172-73
  testing Protect mode, 140
medium model, 52-53, 90, 93-94
megabyte
memory model, 50-54
  huge, 53-54, 90
  large, 53, 90
  medium, 52-53, 90
  small, 48, 51-52
  tiny, 51
message services, 174-79
  binding, 178-79
  class letters, 176
  file format, 176-77
  message format, 175-76
  retrieve and display, 177-78
Mickey, 352, 358-59
MkMsgf utility, 177
MODE command and serial port
  as child process, 403
  DTR=on/off/hs, 394-95, 407-8
  IDSR=on/off, 409
  OCTS=on/off, 410-11
  ODSR=on/off, 410-11
  parameters altered (table), 392
  RTS=on/off/hs/tog, 394-95, 408
  TO=on/off, 397, 406

TO=on/off, 397, 406
uses access mode read, 394
XON=on/off, 409
modem, 382. *See also* RS-232 interface, port
null, 385-86
with DCD signal, 385
module table, 65, 121-25
entry table, 123
imported names table, 123
module header, 122
module reference table, 124
resident names table, 123
searched by DosLoadModule, 132
segment table, 122
money format, 158-60
monitor. *See* device monitor
Monochrome Display Adapter (MDA), 295
attributes, 305-6
fonts not supported, 302
mouse, 17, 32-33, 351-70
device status, 357-58, 365
functions replaced, 367-70
hardware operation, 351-54
in screen group, 18
number of buttons, 357
opening, 354-56
sensitivity in Mickeys, 357-59
test if available, 355
mouse events
flushing, 366
position data, 365
queue, 363-66
reading, 363-65
selection mask, 364
timestamp, 365
mouse monitor, 431-432
data packet, 431
mouse pointer, 33, 353-54
drawing routine, 353, 355-56
excluding, 354, 362-63
position of, 366-67
shape data, 359-62
text mode, 353-54
moving files, 219-20
moving segments
MS-DOS, 17, 21, 95
address space, 21, 23
display modes, 296
display output, 285
executable file, 109, 119-20
use of serial port, 379
MS-DOS screen group, 21
MsgBind utility, 178

named segments, 206
National Language Support (NLS), 158-62
accent keys, 327-28
ASCII collating sequence, 161
case mapping, 161
code page
for keyboard, 327-28, 337
for programs, 161
country info, 158-61
DBCS, 162, 328
keyboard translation, 327, 338-40
near jump and call, 47
near pointer, 46, 245-46
null bytes on serial port, 407
null device, 228-29
detecting, 237
null modem, 385-86
null parameter to DosDevIOCTL, 401-2
null selector, 61
Numeric Processor Extension (NPX)
present in machine, 141
traps raised, 172-73

object file
format, 81
named to Link, 98
object library
named to Link, 99
object module, 73-81
binary image in, 76
bindings in, 76-80
common segments in, 78-79
communal names, 79-80
debug data in, 94-95
defined segments in, 75
dynamic link in, 80, 114-15
essential segments, 74
external names, 78, 114-15
offsets in, 77, 87
program entry point, 81
public names, 77
selectors in, 76-77
offset, 46, 77, 87
open. *See* file, keyboard, mouse open
operating system, 17
operator, 17
ordinals of dynamic links, 108, 116, 134-35

ordinals of dynamic links, 108, 116, 134-35
output. *See* file, device I/O, serial port I/O
overflow trap, 170-71

paging (not done), 199
pallette registers, 303
parameter convention, 42-45, 250
  C language, 44-45
  Pascal language, 44
parent process
  becoming one, 182
  process ID of, 145
Pascal language
  addressing via selectors, 146
  ADS, 136, 146, 246, 402
  ADSPROC, 255
  command parameters, 157
  conventional segments, 89-91
  creating threads, 255-58
  declaring
    directory scan records, 225, 227
    disk info block (BPB), 422-23
    display state parameter, 303
    semaphores, 262
    signal handler, 166
    thread stack, 255
  DGROUP with, 91-94
  DosDevIOCTL, 403
  external names in, 78
  heap use, 93
  indirect calls, 136-37
  linking names in mixed case, 102
  near and far routines, 48
  opening screen as file, 290
  OS/2 version stored by, 141
  program entry point, 81, 129
  public names in, 77
path lists in environment, 156
  searching, 221-22
pathname, 210-12
  "dot" directories, 211
  as parameters, 212
  current directory, 211-12, 217
  drive letter, 210-11
  extracting prefix, 226
  filename, 210
  longest possible, 218-19
  opening a file, 228
  rename and move, 219-20
PC-DOS. *See* MS-DOS
pel. *See* pixel

performance
  arm contention, 244
  file I/O, 216, 243-45
  loading selectors, 63
  segment moving, 197-98
  segment swapping, 199-200
  timer restart, 268
pipe, 188-90
  line delimiters needed, 190
pixel, 31, 286
POINTDD.SYS, 355, 374
pointer, 46
pop-up display, 311-12
Presentation Manager, 285, 312-13
printer, 33
  number of attached, 141
  opening, 228-29
printer monitor, 432-33
  data packet, 433
priority, 28, 251-53
  dynamic, 254
  in local info seg, 153
  query and set, 252-53
  value range, 252
privilege level, 60, 66
  change via call gate, 68, 113-14
  current, 60, 66
  defined to linker, 113
  I/O, 66, 69-70, 113, 318
  in descriptor, 60
  in machine flags, 66-67
  in selector, 61
process, 19-20, 26. *See also* child process
  created by parent, 182-90
  initial registers, 129-30
  initial thread, 128-29
  loading, 125-26
  termination, 130
  using keyboard, 152
process ID (PID), 144-45
  exit list, 173-74
  in local info segment, 153
  of last keyboard user, 152
  of Session Manager, 195
program entry point, 81
program loading, 26
Program Selector. *See* Session Manager
Protect mode, 22, 140
Protect mode addressing, 21, 55-70
public name, 77
  definition file, 115-17

in assembly language, 48
Pascal routines, 48
punctuation of money or dates, 158-60

queue, 30, 280-84
creating, 280-81
discipline of, 281
opening, 281-82
purging, 284
querying, 284
reading, 283-84
scanning, 283
shared storage with, 205, 282
with child session, 192

Read-Only Memory (ROM), 23, 55
read-write pointer
Real mode addressing, 21, 54-55, 83-84, 140
register, 39-50
AX, 41, 49, 129, 132
BP (Base Pointer), 41-45, 129
BX, 41
CS (Code Selector), 20, 46-47, 59, 66, 131, 250
CX, 41, 129
DI (Destination Index), 41-42, 129
DS (Data Selector), 46, 49-50, 62, 94, 132
DX, 41
ES (Extra Selector), 46, 49-50, 129
GDTR (Global Descriptor Table Register), 60
IP (Instruction Pointer), 20, 41-42, 129, 131, 250
LDTR (Local Descriptor Table), 20, 46, 62
SI (Source Index), 41-42
SP (Stack Pointer), 20, 41-45, 129, 131, 250
SS (Stack Selector), 20, 46-47, 94, 129, 131, 250
registering replacement functions
display functions, 321-24
renaming files, 219-20
response file, 96-97
root directory
RS-232 interface, 380-91
break signal, 387, 415
connector, 381
DCE uses female, 381
DTE uses male, 381
data format, 387-89
DCE (Data Communication Equipment), 381-83
DTE (Data Terminal Equipment), 381-83
parity, 388
stop bit, 388
transmission rate, 389
RS-232 signal lines, 383-86
CTS (Clear To Send), 384
for output flow control, 391
in null modem, 386
data lines 2 and 3, 383
DCD (Data Carrier Detect), 384-85
for output flow control, 391
DSR (Data Set Ready), 384
for output flow control, 391
DTR (Data Terminal Ready), 383
at close time, 396
at open time, 394-95
direct control, 416
for input flow control, 390
policy set, 407-8
mark and space levels, 387
RI (Ring Indicator), 385
RLSD (Received Line Signal Detect), 384-85
RTS (Request to Send), 384
at close time, 396
at open time, 394-95
direct control, 416
in null modem, 386
policy set, 408
summary table, 382
runtime library, 73
default, 81
defines entry point, 81
defines stack, 93

save and restore display, 313-14
scan code
input, 335
tables, 344-50
scheduling, 28
algorithm, 253-54
max wait value, 254
min, max timeslices, 254
parameters in info segment, 151
priority values, 252
use of dispatch classes, 252-53
screen. *See* display
screen group, 18, 20. *See also* session
created by session manager, 19, 190-91
defined 18

foreground vs. background 18
number of foreground, 151
number of own, 153-54
number that exist, 151
pop-up display is, 311
switching, 288
sector size, 218, 422-24
segment (as linked)
alignment type, 83-84, 91
class, 84, 89, 91
BEGDATA, 90
BSS, 92
CODE, 89
ENDCODE, 90
FAR, 90
STACK, 93
combine type, 84-85, 91
common, 78-79
defined, 75
DOS order, 88-89
essential, 74
grouping, 87
name, 89
packing, 87-88, 105
segment (of storage), 46
code, 29, 47
conforming, 67, 113-14
data, 29, 49, 110-11
discardable, 198
disposable, 203-4
huge, 53-54, 90, 201-3, 207
named, 206
shared, 29, 205-7
size, 57
stack, 47, 57
task state, 69
segment alignment, 83-84
segment descriptor, 22, 56-60
accessed bit, 58-59, 198
address, 58
call gate, 68, 113-16
code/data bit, 59, 196
conforming, 67-68, 113-14, 196
present bit, 58-59, 198
privilege level, 60, 113-14
readable/read-only bit, 59, 112-13
size, 57, 111
segment fault, 200
segment moving, 197-98
segment offset, 46
segment selector, 46-50
all-zero is null, 61
format, 61-62
in Real mode, 54-55
loading into register, 62-63, 76
shared segments, 64-65
with huge segments, 53-54
segment swapping, 23, 58-59, 127-28, 198-200
when locking display buffer, 316-17
segments used by compilers, 89-91
semaphore, 29-30, 261-68
as key, 262-63, 271
as signal, 265-66, 271
fundamental operations, 261
local ("RAM"), 29
system, 30, 266-67
serial port, 34, 379-418
data format, 387-89
flow control, 390-91
input, 409
output, 410-11
input queue, 396, 411
line protocol, 403-4
number of attached, 141, 379
opening, 229
overrun, 389
parity, 388
Real vs. Protect modes, 379-80
status queries, 411-15
stop bits, 388
transmission rate, 389, 403
transmit queue, 396, 411
serial port I/O, 379-418. *See also* I/O Control, RS-232
break output, 415
closing, 393-95
DCB (Device Control Block), 404-10
discarding nulls, 407
DTR policy, 394-95, 407-8
errors, 400-1, 407
opening, 393-95
reading, 398-401
RTS policy, 394-95
summary tables, 392, 402
timeout policies
for read, 398-99
for write, 397
query and set, 406
writing, 396-98
session, 19, 190-95
binding, 194-95

creating, 191-94
switching, 194-95
Session Manager, 19, 190-91
hot key, 191
keyboard LEDs, 327
menu line text, 191-92
mouse buttons, 357
privileges of, 195-96
shared storage, 29, 64-66, 205-7
by Getting, 206, 282
by Giving, 207
code segments, 29, 65
dynamic link data segments, 29
huge segments, 207
named segments, 206
queues, 205, 282
shift states, 327, 336-37
bitset diagrams, 336
reading from keyboard, 332-33, 335
signal, 163-68. *See also* trap
acknowledging, 167
blocking, 166
control-C and -Break, 27, 163-68
handler, 27, 164, 166-68
holding, 164
ignoring, 165-66
information, 27
termination, 163-68
versus trap, 169
signal handler
small model, 51-52, 93-94
stack, 38, 46
of privileged code, 69
size of, 93, 105, 110
stack frame, 42-43
stack segment not used, 47, 57
standard file handles, 26, 186
alias created, 187-88
in file table, 214
pipe substituted for, 188-90
print handle not standard, 214
set up by parent, 26, 186-90
standard output
checking for screen, 289
storage allocation, 23, 200-7
and local descriptors, 202
and segment moving, 197
and segment swapping, 198-200
disposable segments, 203-4
huge segments, 201-2
performance of, 197, 199
shared segments, 205-7
suballocation, 203
storage available, 201
storage reserve, 199
store a register, 40
stub loader, 24
suballocation, 203
subdirectory. *See* directory
subroutine linkage, 42-45
swapping segments. *See* segment swapping
system semaphore, 30, 266-67

task state segment, 69
termination signal
text mode display, 31
thread, 20, 26, 249-58
creating, 250-51
defined 20
ending, 251
first of a process, 128-29
priority, 251-53
registers of, 69, 129, 249, 251
resuming, 251
suspending, 251
watchdog, 32, 318-20
thread ID (TID)
time and date, 30, 142-43, 146. *See also* file date and time
from CLOCK$, 239
time zone, 142-44
time-slicing. *See* scheduling
timer, 30, 268-69
resolution, 147
restart delay, 268
services, 268-69
used for time-slicing, 28
timestamps, 30
in global information segment, 147-50
long-term, 147
short-term, 149-50
trace flags, 152
trap, 168-73
array bounds, 171
divide by zero, 170
handler, 27, 169-73
invalid opcode, 171
list of traps, 168
numeric processor, 172-73
overflow (INTO), 170-71
segment fault, 200
versus signal, 169

triple fault, 22

uppercase conversion, 161
user. *See* operator

verify switch. *See* write-verify switch
version number, 140-41, 146
video display. *See* display
Video Graphics Array (VGA), 287, 295
  pallette registers, 303, 317-18
volume label. *See* disk label

watchdog thread, 32, 318-21
whole-disk I/O. *See* disk volume I/O
word, 38
write-verify switch
  belongs to process, 19
  query and set, 215-16

XON and XOFF, 390-91, 395, 409-11, 417-

System Calls

Here are the OS/2 system calls, described in alphabetical order. Each call's operation is stated in terse, technical prose. For a broader explanation, and for context, and for the relationships among groups of calls, follow the *discussion* headings to topics in the first half of the book.

With each call is listed the error codes that I've been able to elicit from it with their causes. If you discover other error codes or other causes, by all means notify me at the address in the *Preface*.

A call's invocation is shown in three forms: as a diagram of parameters on the machine stack when the call is made, and as declarations in C and Pascal. When an address must be passed on the stack, the diagram reflects this with an arrow pointing to a diagram of the addressed object. A broad arrow shows that the object can be modified during the call; a narrow one shows that the object is only input to the call.

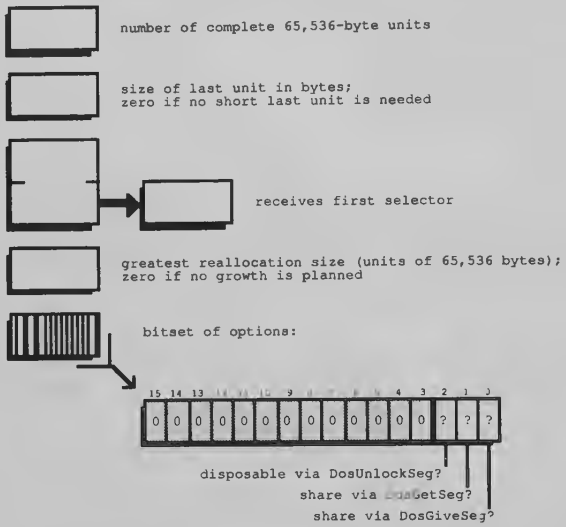
The C and Pascal declarations, and declarations of the data types and constants they need, are available as include files in the Disk Appendix to this book. The *include* headings give the name of the file to include, so that *Include: STORAGE* means a C program would include STORAGE.H and a Pascal program, STORAGE.PIN from the Disk Appendix.

For those who would rather write their own, the data types named in the function declarations have been reprinted in *Appendix A: Collected Data Structures*.

In order to avoid any question of copyright ownership, I developed these data types and constants without reference to the ones defined in the "Programmer Toolkit" packages available from IBM and Microsoft. As a result, the names in Appendix A often differ from those used in example code from other sources, and the include files in the Disk Appendix are organized differently and have different names from the ones sold by the vendors of OS/2. I apologize for any confusion this causes.

DosAllocHuge

Allocate storage over 64K bytes



A contiguous block of storage is allocated of the requested size and added to the address space of the current process as a series of segments. Entries are added to the process's local descriptor table for each unit of 64K bytes, including the short last unit and any segments that might be needed for later growth. The entries are regularly spaced so that there is an arithmetic relation between selectors for adjacent 64K units. Only the first selector is returned. The other selectors may be derived from it via the Huge Shift, a constant returned by *DosGetHugeShift* and also available in the system data returned by *DosGetInfoSeg*. The difference between selectors for adjacent units is 0001h shifted left by the Huge Shift. All the selectors may be developed at once and saved in an array. Alternatively a desired selector can be computed on the fly by taking the ordinal number of a unit, shifting it left by the Huge Shift, and adding it to the first selector.

Discussion: *See* 7.3.3 **Include:** STORAGE

Related: *DosAllocSeg, DosFree, DosGiveSeg*

Family API: size of last unit is rounded up to 16-byte multiple; first selector is physical segment address; share option is ignored; the most segments that can be allocated is 8 (640K bytes less segments for DOS and the program)

Errors to consider:

8 Not enough vacant storage or no more local selectors

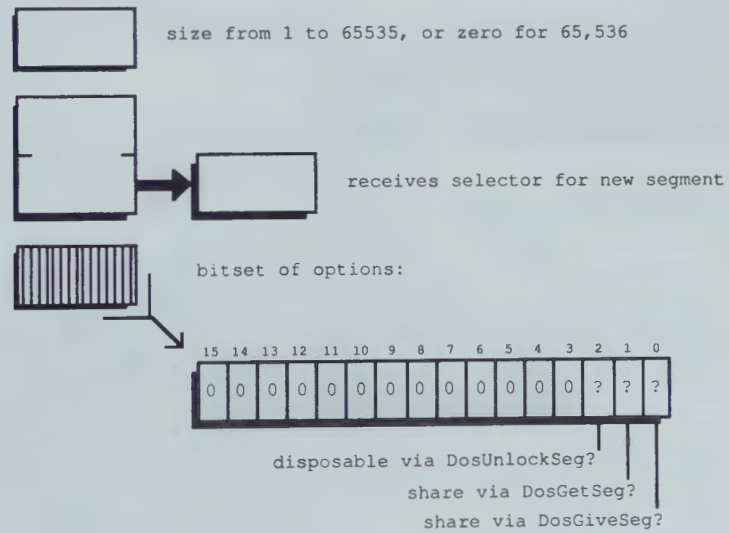
87 The share option must not exceed 07h.

```
extern unsigned far pascal DOSALLOCHUGE(
```

```
                        unsigned,
                        unsigned,
                        SELECTOR far *,
                        unsigned,
                        unsigned);
#define DosAllocHuge DOSALLOCHUGE

function DosAllocHuge  (     NumSeg: word;
                             EndLen: word;
                        vars TheSel: Selector; { usually an ADS.S }
                             MaxSeg: word;
                             Option: word
                        ) : word; external;
```

DosAllocSeg

Allocate storage to 64K



A block of storage of the requested size is allocated and added to the address space of this process as a unique segment. The selector to it is returned. The options may be used in any combination. Bit 0 makes the new segment elegible to be shared by Giving with *DosGiveSeg*. Bit 1 makes it elegible to be shared by Getting with *DosGetSeg*. Bit 2 makes it possible to mark the segment disposable with *DosUnlockSeg*.

Discussion: *See* 7.3.2 **Include:** STORAGE

Related: *DosAllocHuge, DosFreeSeg, DosSubSet, DosUnLockSeg, DosGiveSeg, DosGetSeg*

Family API: requested size is rounded up to a multiple of 16 bytes; the selector returned is the physical segment address; all options are ignored as none apply under MS-DOS

Errors to consider:

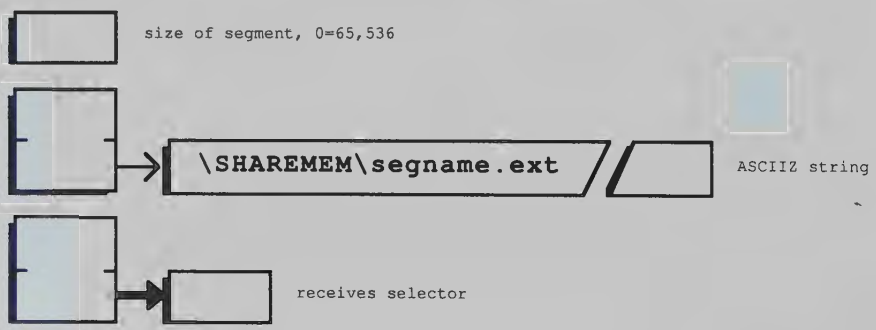
8 Not enough vacant storage or no more local selectors

87 The option word may not exceed 0007h

```
extern unsigned far pascal DOSALLOCSEG(
                        unsigned,
                        SELECTOR far *,
                        OPTIONSET);
#define DosAllocSeg DOSALLOCSEG

function DosAllocSeg    (       size: word;
                        vars TheSel: Selector;
                            TheOpts: OptionSet
                        ) : word; external;
```

DosAllocShrSeg

Allocate named storage segment



A block of storage of the requested size is allocated and added to the address space of the current process as a unique segment. The selector to it is returned. In addition, the name-string is associated with the segment in a table maintained by the system. Other processes may gain access to the segment using *DosOpenSeg*. The system keeps a count of processes that have access to the segment, and retains the segment until none do.

Discussion: *See* 7.4.1 **Include:** STORAGE

Related: *DosAllocSeg, DosOpenSeg, DosFreeSeg*

Family API: not available
Errors to consider:
8 Not enough vacant storage or no local selectors
123 The segment name must be an ASCIIZ string of the form; "\SHAREMEM\segname.ext" and use characters valid in filenames
183 A segment of that name already exists

```
extern unsigned far pascal DOSALLOCSHRSEG(
                        unsigned,
                        char far *,
                        SELECTOR far *);
#define DosAllocShrSeg DOSALLOCSHRSEG

function DosAllocShrSeg(     TheSize : word;
                        vars TheName : char; { e.g. Astring[1] }
                        vars TheSelector: Selector
                        ) : word; external;
```

DosBeep

Sound speaker

pitch in Hertz, from 37 HZ to 32,767 Hz

duration in milliseconds

If the pitch is less than 37 Hz or greater than 32,767 Hz (a binary value less than 0025h or greater than 7fffh) no sound is delivered and the function returns immediately with an error code. Otherwise a tone of the specified pitch and duration is sounded through the speaker in the system unit. Use of the speaker is serialized so that only one sound at a time is issued. If another sound is in progress when this call is made, the calling thread is suspended until the speaker is free. If more than one thread waits for the speaker it is unpredictable which will make its sound first. The system does not return from the call until the sound has been made for its full duration. However, the sound will be cut off if the calling process is terminated (by a Control-Break signal for example).

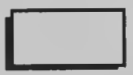
Discussion: none **Include:** SPEAKER
Related: none
Family API: available
Errors to consider:
395 The pitch value must be between 32 and 32,767

```
extern unsigned far pascal DOSBEEP(int, unsigned);
```

```
#define DosBeep DOSBEEP

function DosBeep(FreqHz : integer;
                 Duration : word):word; external;
```

DosBufReset

Force output to disk



handle of file

If any output data has been buffered and not written for this file, it is written to disk. Disk directory information is updated to disk, ensuring that all space allocated to the file is permanently recorded.

Discussion: *See* 8.11.6 **Include:** FILES
Related: *DosClose, DosWrite*
Family API: not available
Errors to consider:
6 The handle must be open to a disk file

```
extern unsigned far pascal DOSBUFRESET(HANDLE);
#define DosBufReset  DOSBUFRESET

function DosBufReset   (TheHandle : Handle
                        ) : word; external;
```

DosCLIAccess

Ask permission to block interrupts

(no parameters)

The call should be contained in a code segment with I/O privilege (linked with the IOPL keyword). The program requests the right to use the CLI (clear interrupt mask, preventing interrupts) and STI (set interrupt mask, allowing interrupts) instructions. This call is implicit in a *DosPortAccess* call and need not be used if that call is used. In present levels of OS/2 running in 80286, I/O privilege is a blanket right that is conferred automatically on code linked with the IOPL keyword; the hardware lacks the ability to discriminate between kinds of I/O-related instructions. Thus in present systems this call is only a formality.

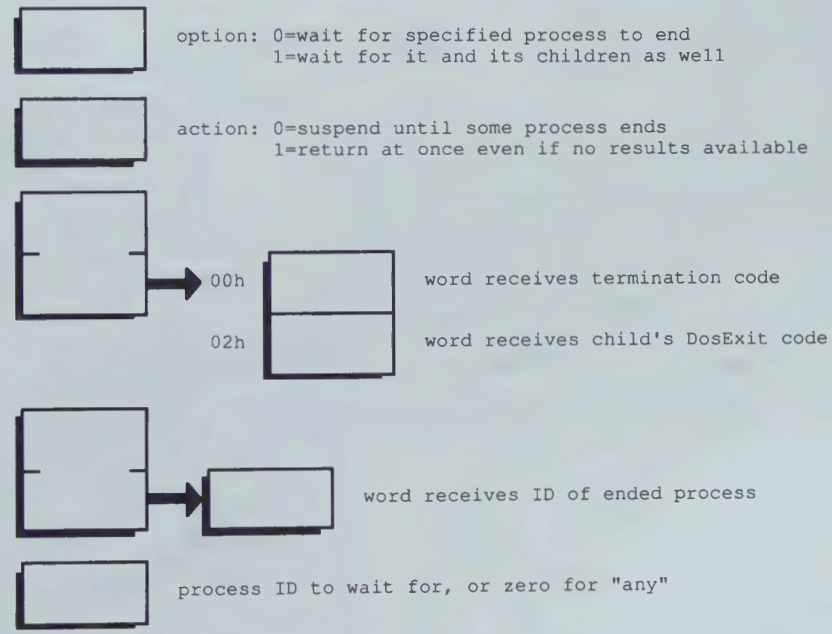
However, in future levels of the system on more elaborate hardware, the system may have the ability to enforce the requirement. Since the call takes almost no time to execute, use it now in order to be portable later.

Discussion: none **Include:** DEVICE
Related: *DosPortAccess*
Family API: available
Errors to consider: none

```
extern void far pascal DOSCLIACCESS(void);
#define DosCLIAccess DOSCLIACCESS

procedure DosCLIAccess; external;
```

DosCWait

Wait for child processes to end



When the last parameter is nonzero this call samples the state of the process it designates. If it does not exist as a child of the current process an error code is returned. If it was started with *DosExecPgm* option 2 and has since ended, its saved termination and exit codes are returned in the third parameter. If it is still active, the second parameter is consulted: if that is 1, control returns at once with an error code. Otherwise the current process is suspended until the designated

process (and its children, if the first parameter is 1) has ended; then the termination and exit codes are set and control returns.

When the last parameter is zero the call polls for any ended child process. If no children exist, an error code is returned. If a child started with *DosExecPgm* option 2 has ended, its saved termination and exit codes are returned in the third parameter. Otherwise one or more children exist; the second parameter is consulted. If it is 1, control returns with an error code. Otherwise the current process is suspended until any child process (or every child process, if the first parameter is 1) has ended. Termination and exit codes are returned only after the end of processes that were started with *DosExecPgm* option 2. The termination codes are

0 = normal DosExit, see exit code
1 = ended by hard I/O error
2 = ended by hardware trap
3 = ended by external signal (*DosKillProcess*)

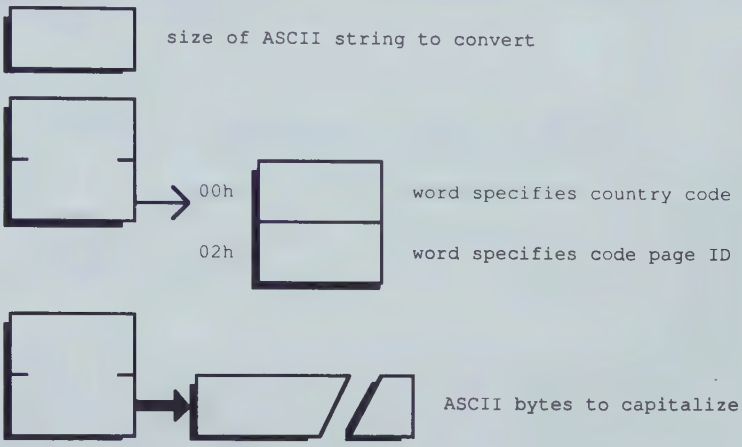
The exit code returned is only significant if the termination code is zero; then it is the argument given with the final *DosExit* call.

Discussion: *See* 6.1.2.1.2 **Include:** SESSION
Related: *DosExecPgm, DosKillProcess*
Family API: not available
Errors to consider:
13 The first and second parameters must be zero or 1
128 (last parameter nonzero) The designated process is not a child of this one
128 (last parameter zero) The current process has no remaining children
129 (last parameter zero, second parameter 1) all children are active

```
extern unsigned far pascal DOSCWAIT(
                        int,            /* action code */
                        int,            /* wait option */
                        long far *,     /* results */
                        PROCESSID far *, /* who ended */
                        PROCESSID);     /* who to wait for */

function DosCWait(      ActionCode : integer;
                        OptionCode : integer;
                    vars Resultwords: word; { TWO words! }
                    vars WhoEnded   : ProcessID;
                         WhoToWaitOn: ProcessID
                         ):word; external;
```

DosCaseMap

Capitalize ASCII using national language code



The first word of the second parameter specifies a country number (see Table 5.2, p. 160 or zero to indicate the country set in the configuration file. The second word specifies a code page (see Table 5.3, p. 162), or zero to indicate the primary code page for the specified country. The ASCII characters in the buffer described by the first and third parameters are converted in place to uppercase letters under the rules of the specified code page.

Discussion: *See* 5.2, 5.2.2 **Include:** SYSINFO

Related: *DosGetCollate, DosGetCtryInfo, DosGetCP*

Family API: available.

Errors to consider:

396 Couldn't locate the COUNTRY.SYS file

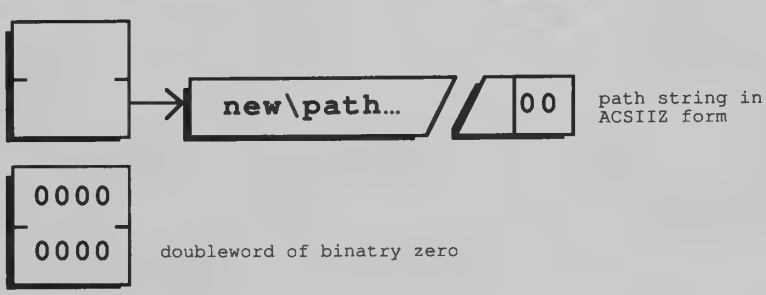
397 Couldn't open the COUNTRY.SYS file

398 The country number must be one supported by OS/2

```
extern unsigned far pascal DOSCASEMAP(
                        unsigned,
                        struct CTRYCPID far *,
                        char far *);
#define DosCaseMap DOSCASEMAP

function DosCaseMap  (     BufSize : integer;
                       vars TheCtry : CtryCPID;
                       vars Buffer  : char { e.g. AsciiChars[17] }
                         ) : word; external;
```

DosChDir

Set current directory path



The current directory path is set for some drive. The operation applies to the current drive unless the path string specifies a drive-letter. If the new path string doesn't begin with the root, the system prefixes the current directory path to it. The total new path string, exclusive of drive-letter, colon, and null terminator, may not exceed 64 bytes. The system checks out the path. If all members can be found and are directories, the path string is set.

Discussion: *See* 8.5.3, 8.5.2, 8.1.4 **Include:** FILES

Related: *DosQCurDir, DosSelectDisk*

Family API: available

Errors to consider:

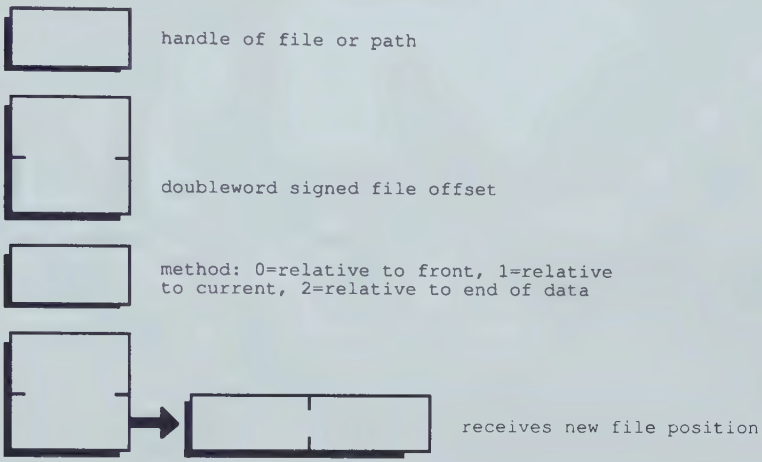
3 Path not found: invalid drive or a member doesn't exist or isn't a directory or the total path string exceeds 64 bytes

5 The last file named in the path isn't a directory

```
extern unsigned far pascal DOSCHDIR(char far *, long);
#define DosChDir DOSCHDIR

function DosChDir       (vars ThePath : char; { e.g. PathStr[1] }
                             AnyZero : integer4 { must be zero }
                        ) : word; external;
```

DosChgFilePtr

Set file read-write pointer



If the file is a device, an error is returned. If the file is a disk file, its read-write position is altered in one of three ways depending on the method parameter. If Method is 0, the passed offset becomes the new read-write pointer. If Method is 2, the passed offset is added to the end of data value to make the new file pointer. If Method is 1, the passed offset is added to the current read-write pointer to make the new pointer. The new pointer is returned.

Discussion: *See* 8.11.5 **Include:** FILES

Related: *DosOpen*

Family API: available

Errors to consider:

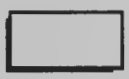
6	The handle must be open to a disk file
87	The method number must be 0, 1 or 2
131	With these parameters the file pointer would end up negative
132	Devices cannot have pointers changed

```
extern unsigned far pascal DOSCHGFILEPTR(
                        HANDLE,
                        FILEOFFSET,
                        int,
                        FILEOFFSET far *
                                );
#define DosChgFilePtr DOSCHGFILEPTR
```

```
type FileOffset = integer4;
const
     FP_Front = 0;
     FP_Back = 2;
     FP_Relative = 1;
function DosChgFilePtr (    TheHandle : Handle;
                            ChgOffset : FileOffset;
                            TheMethod : integer;
                       vars NewOffset : FileOffset
                        ) : word; external;
```

DosClose

Close file handle



handle of file or pipe

The association between the file and the handle number in this process is severed. If the file is a disk file used for output, buffered output is written and the disk directory is updated. If the file is a device, the device driver is notified of the close and may or may not take action. If the print spooler has been diverting print output to disk it will close its file of spooled output.

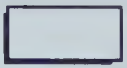
If the same file has been tied to other handles in this process via *DosDupHandle*, the other handles are not affected; they are still open to that file. If the file handle was inherited by a child process, that process's handle is still valid and open to the file.

Discussion: *See* 8.8 **Include:** FILES
Related: *DosOpen*
Family API: available
Errors to consider:
6 The handle must be one returned by *DosOpen* and not yet closed

```
extern unsigned far pascal DOSCLOSE(HANDLE);
#define DosClose DOSCLOSE

function DosClose       (TheHandle: Handle
                         ) : word; external;
```

DosCloseQueue

Give up access to queue



handle returned by DosOpenQueue
or DosCreateQueue

The current process gives up access to the specified queue. If the current process created the queue, the system discards the queue and any elements in it. Other processes that have opened the queue will, if they try to write into it, get an error code.

Discussion: *See* 9.7 **Include:** QUEUES
Related: *DosCreateQueue*, *DosOpenQueue*
Family API: not available
Errors to consider:
337 The handle must be for an existing, opened queue

```
extern unsigned far pascal DOSCLOSEQUEUE(HANDLE);
#define DosCloseQueue DOSCLOSEQUEUE

function DosCloseQueue( TheHdle : Handle
                      ):word; external;
```

DosCloseSem

Give up access to system semaphore

32-bit handle of system semaphore

Give up access to a semaphore gotten with *DosOpenSem* or *DosCreateSem*. When no process has access to it, the semaphore will be discarded. This call will be done automatically as part of terminating a process, but it is good practice to close all semaphores explicitly.

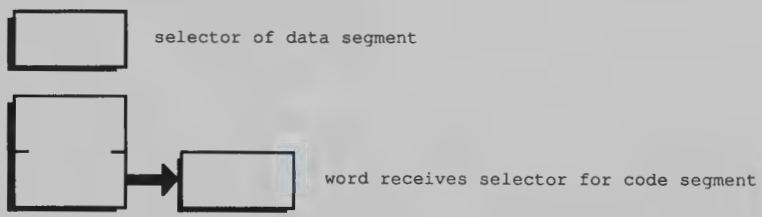
If this process owns the semaphore (has set or claimed it and not cleared it), error 102 will be returned (but the semaphore is closed regardless). If another process was waiting to own the semaphore it will be released but it will receive error 105. Avoid this by clearing the semaphore before closing it.

Discussion: *See* 9.4.5 **Include:** ASYNC
Related: *DosCreateSem, DosOpenSem*
Family API: not available
Errors to consider:
6 The handle must be one returned by *DosOpenSem* or *DosCreateSem*
102 This process owned the semaphore and didn't clear it

```
extern unsigned far pascal DOSCLOSESEM(SEMHANDLE);
#define DosCloseSem DOSCLOSESEM

function DosCloseSem    (TheHdle:SemHandle
                         ):word; external;
```

DosCreateCSAlias

Treat data segment as code



The data segment specified by the first parameter must be writable, private to the current process (not shareable with other processes) and must not be a segment allocated by *DosAllocHuge*. It is presumed that executable code has been (or will be) placed in the segment. A new entry is created in the Local Descriptor Table of the current process that describes the segment as a code segment, and a selector for that descriptor is returned in the second parameter. That word may be used as the selector portion of a far pointer used in a call or jump instruction. The descriptor specifies the segment as "conformable," meaning it may be called from a normal segment or from one with I/O privilege (in order to execute I/O instructions, the caller must have I/O privilege). Both selectors remain usable, so the segment contents may be updated as data or executed as code alternately (it is up to the application to avoid altering code while it is being executed). Either selector may be freed with a call to *DosFreeSeg*; the other remains in force.

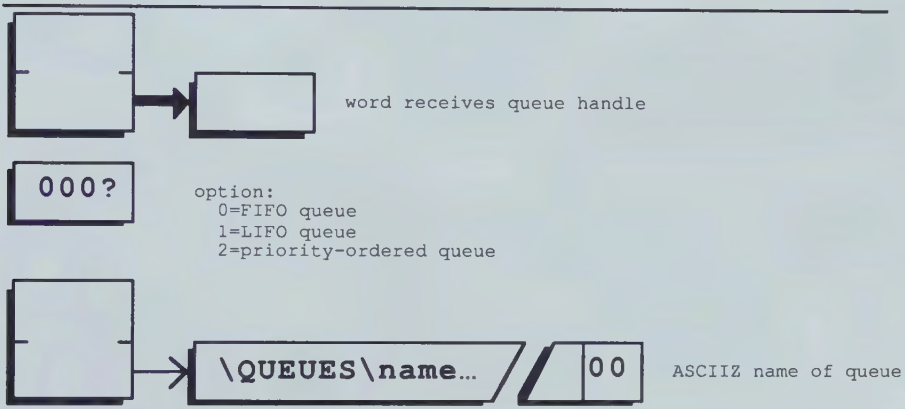
Discussion: *See* 6.4 **Include:** SESSION
Related: *DosFreeSeg*
Family API: available
Errors to consider:

5 The first parameter must select a private, read-write data segment

```
extern unsigned far pascal DOSCREATECSALIAS(
                                SELECTOR,
                                SELECTOR far *);
#define DosCreateCSAlias DOSCREATECSALIAS

function DosCreateCSAlias ( DataSel : Selector;
                        vars CodeSel : Selector
                        ) : word; external;
```

DosCreateQueue

Create queue



The third parameter specifies a name in the form of a filename in an imaginary directory "QUEUES"; that is, the name must begin with "\QUEUES\" and continue with a properly formatted filename and an optional file extension. If a queue of the specified name exists, an error is returned; otherwise the queue is created. The current process owns the queue, which means that only this process may use the *DosReadQueue* and *DosPeekQueue* calls on it, and that it will cease to exist when this process closes it or terminates. Other processes may get access to the queue with *DosOpenQueue* and may put data into it with *DosWriteQueue*.

Discussion: *See* 9.7 **Include:** QUEUES

Related: *DosCloseQueue, DosReadQueue, DosOpenQueue, DosPeekQueue, DosWriteQueue*

Family API: not available

Errors to consider:

332 A queue of that name already exists and may not be created

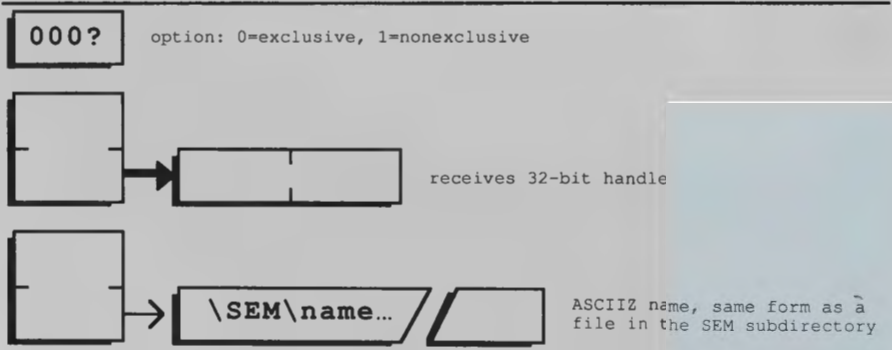
335 The queue name must start with "\QUEUES\" and continue with a properly formed filename and optional extension

336 The queue-discipline number must be zero, 1 or 2

```
/extern unsigned far pascal DOSCREATEQUEUE(
                        HANDLE far *,
                        int,
                        char far *);
#define DosCreateQueue DOSCREATEQUEUE

function DosCreateQueue(  vars TheHdle : Handle;
                               TheOption : integer;
                          consts TheName : char { e.g. QNameStr[1] }
                          ):word; external;
```

DosCreateSem

Create system semaphore



If no semaphore of this name exists, one is created and a 32-bit handle is returned (note that the more common *file* handle has only 16 bits). The handle may be used in any semaphore operation by the active process, and may be passed to a child process as well.

Any process may use *DosOpenSem* to access the semaphore, so it can be used to coordinate multiple asynchronous programs. A semaphore has a reference count which is set to 1 when it is created, incremented for each process that opens it, and decremented for each process that closes it or terminates after opening it. The semaphore will exist as long as its reference count is above zero.

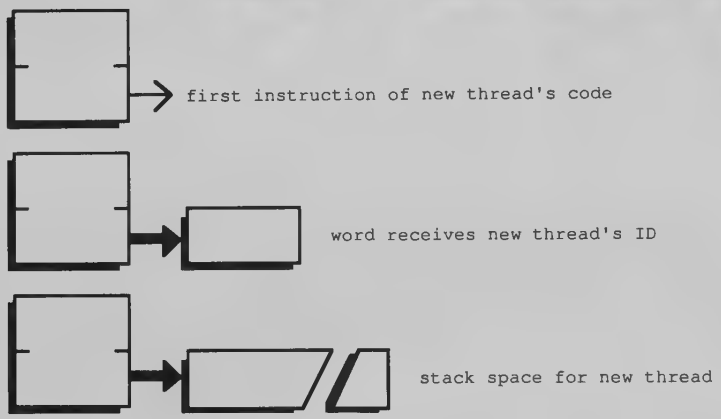
The process that claims a semaphore with *DosSemRequest* or sets it with *DosSemSet* is its owner. An "exclusive" semaphore, once owned, may not be cleared or set by any other process until its owner releases it with *DosSemClear*. A "nonexclusive" semaphore can be cleared or set by any process at any time.

Discussion: *See* 9.4.5 **Include:** ASYNC
Related: *DosCloseSem, DosOpenSem*
Family API: not available
Errors to consider:
87 The option must be 0 or 1
100 The system limit on open semaphores has been exceeded
123 The name must be an ASCIIZ string of the form "\SEM\filename.ext" using only characters valid in a filename
183 That semaphore already exists (open it instead)

```
extern unsigned far pascal DOSCREATESEM(int,
                                        SEMHANDLE far *,
                                        char far *);

#define DosCreateSem DOSCREATESEM

function DosCreateSem  (     UseType: integer;
                        vars TheHdle: SemHandle;
                        vars TheName: char { e.g. AString[1] }
                        ) : word; external;
```

DosCreateThread

Create asynchronous thread



A new thread is created. Its address space is that of the current process. Its CS:IP registers are set from the first parameter. Its SS:SP registers are set from the third parameter. Since a stack push decrements SP and then stores, the third parameter should consist of the segment selector and the highest possible offset in the stack space.

The rest of the new thread's registers are those of the calling thread. Its priority is that of the calling thread. It has full access to all storage, files, system semaphores and other resources owned by the current process. The thread will execute independently and asynchronously until it makes a *DosExit* call. The far address of a *DosExit(0,0)* call is pushed onto the stack space so that a far return will terminate the new thread.

The maximum number of threads in the system is set by the *threads=* statement in the system configuration file. There may be other, lower limits on the number of threads per process.

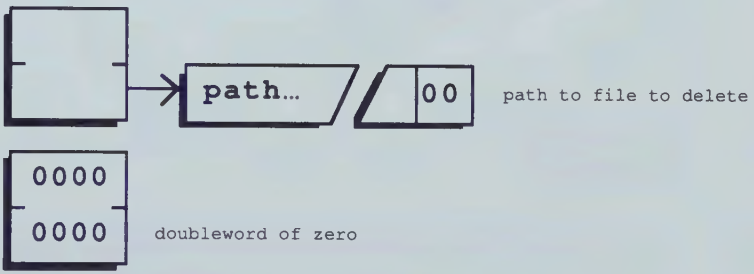
Discussion: *See* 9.1.2 **Include**: ASYNC
Related: *DosExit*
Family API: not available
Errors to consider:
155 The system can't create any more threads in this process

```
extern unsigned far pascal DOSCREATETHREAD(
                        ADSPROC, /* typedef of void (far *)() */
```

```
                        THREADID far *,
                        ADSMEM); /* typedef of void far * */
#define DosCreateThread DOSCREATETHREAD

function DosCreateThread( TheCode : ADSPROC;
                          vars TheID : ThreadID;
                           TheStack: ADSMEM
                         ) : word; external;
```

DosDelete

Delete a file



If the specified file does not have the read-only attribute and is not in use by another process, it is deleted.

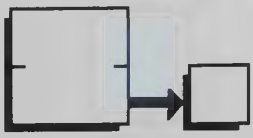
Discussion: *See* 8.6.3 **Include:** FILES
Related: *DosMove*
Family API: available
Errors to consider:

2 File not found (but path is OK; file may be a directory)
3 Path not found: invalid drive or a member doesn't exist or isn't a directory
5 Access denied (file is read-only)
32 Sharing violation (file is open in another process)

```
extern unsigned far pascal DOSDELETE(char far *, long);
#define DosDelete DOSDELETE

function DosDelete      (vars ThePath : char;
                             AnyZero : integer4
                         ) : word; external;
```

DosDevConfig

Query attached devices



byte receives answer

inquiry number:
0=number of printers
1=number of serial ports
2=number of diskette drives
3=math coprocessor attached?
4=system sub-model number
5=system model number
6=display adapter type

word of zero

Information about the attached system hardware is returned as a count or flag byte in the byte addressed by the first parameter. The response to inquiry 3, math coprocessor attached, is zero for "no" and nonzero for "yes." The response to inquiry 6, display adapter type, is zero for a monochrome display adapter and nonzero for another monitor type. The response may or may not reflect the difference between CGA, EGA and VGA color adapters; use *VioGetMode* to find out the screen dimensions in characters and pixels.

Discussion: *See* 5.1.1.3 **Include:** SYSINFO

Related: none

Family API: available

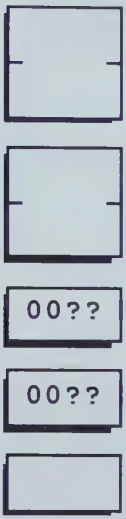
Errors to consider:

87 The inquiry number must be between 0 and 6 and the last parameter must be a word of zero.

```
extern unsigned far pascal DOSDEVCONFIG(
                                char far *, /* answer */
                                int,    /* query number */
                                int); /* must be zero */
#define DosDevConfig DOSDEVCONFIG

function DosDevConfig  (vars Answer: byte;
                        QueryNo : integer;
                        AnyZero : integer
                        ) : word; external;
```

DosDevIOCTL

Request service from a device driver



If the device driver that is associated with the given file handle does not accept I/O control calls, an error code is returned. Otherwise the first four parameters are passed directly to the device driver for interpretation. The device driver's return code is returned as the result of the operation. Different device drivers support different sets of category and function codes. The use of the first and second parameters depends on the category and function codes and on the rules of the particular device driver. For some functions, some device drivers require one or both of these parameters to be null pointers, that is, doublewords of zero. In other cases the addressed "packet" may be a byte, a word, a doubleword or a buffer area of some size. In general the parameter packet will contain control or descriptive information while the data packet will contain or receive the data. The category codes are summarized in Table 13.1, p. 376 The function codes for category 1, serial port, are summarized in Table 14.3, p. 402. A selection of functions in category 8, physical disk operations, is summarized in Table 15.1, p. 419.

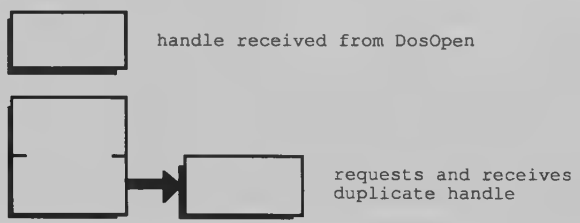
Discussion: See 13.4, 13.5, 14.3.4 **Include:** DEVICE
Related: *DosOpen*
Family API: not available
Errors to consider:

22 The category and function codes must be ones the driver supports
31 This "general failure" code is returned by the COM driver for almost any miscoding of parameter packet or data packet parameters

```
extern unsigned far pascal DOSDEVIOCTL(
                  void far *,      /* data packet */
                  void far *,      /* parameter packet */
                  int,             /* device function code */
                  int,             /* device category */
                  HANDLE);         /* handle */
#define DosDevIOCTL DOSDEVIOCTL

{ use NilADS constant for null packets in category 1 calls }
function DosDevIOCTL(   DataPacket : ADSMEM; { ADS of data packet }
                        ReqParm :  ADSMEM;   { ADS of request packet }
                        FuncCode : integer;  { generic function code }
                        CatCode : integer;   { generic category code }
                        TheHdl : Handle      { handle from
DosOpen/DosMonOpen}
                        ):word; external;
```

DosDupHandle

Create duplicate handle



If the second word contains -1 (0ffffh), a new handle is allocated and stored in the word. Otherwise the second word must contain a handle that is presently open; it is closed as by *DosClose*. Then the second handle is associated with the identical file state as the first file.

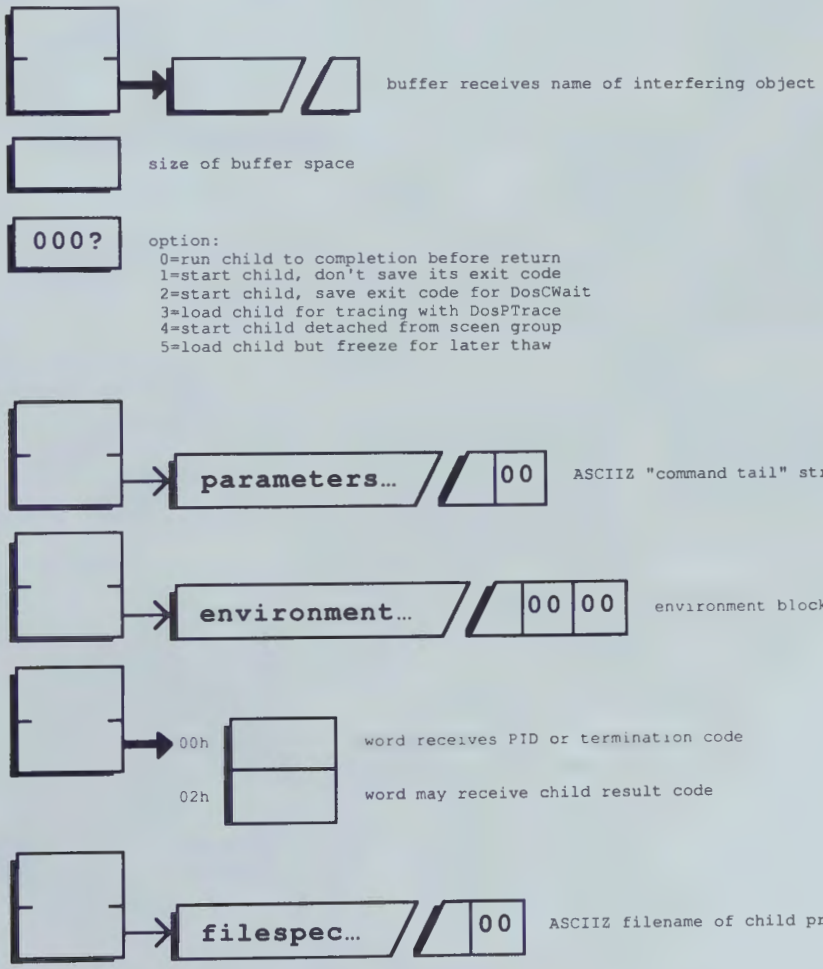
Discussion: *See* 8.10.3, 6.1.5.2 **Include:** FILES
Related: *DosOpen, DosQFHandState, DosQHandType, DosSetMaxFH*
Family API: available
Errors to consider:
4 No available handles
6 One of the handles is not open

```
extern unsigned far pascal DOSDUPHANDLE(HANDLE, HANDLE far *);
#define DosDupHandle DOSDUPHANDLE
```

```
function DosDupHandle  (OldHdl : Handle; vars NewHdl : Handle
                       ) : word; external;
```

DosExecPgm

Initiate child process



The filename from the last parameter is looked up in the table of loaded modules. If it appears, that loaded program is used. Otherwise the final parameter must be

the filename and extension of an executable file. If the parameter does not include a path, the PATH list in the environment of the current process is searched as by *DosSearchPath*, and the path to the first matching filename and extension is used. The completed filespec is opened as by *DosOpen* (read-only, deny-write, no inheritance). If the file can be opened and has the internal format of an executable program, it is loaded into storage and its bindings are resolved.

Once a loaded program exists, an environment is created from the fourth and fifth parameters. The fifth parameter may be a doubleword of zero, indicating that the current process's environment is to be used. Otherwise the parameter must address a block of zero or more ASCIIZ strings followed by a null byte (one byte of zero comprises the minimal environment). The fourth parameter may be a doubleword of zero, indicating that there are no arguments; otherwise it addresses zero or more argument strings in exactly the same format as the environment. By convention, an argument block should contain two strings, the first being the name of the executable file (same as the last parameter) and the second being a "command tail." The final environment segment will contain both blocks of strings. Its selector will be passed to the new process in AX, the offset of the arguments in BX.

The new process inherits a table of open files which contain copies of all inheritable handles owned by the current process. A single thread is created to animate the new process. Now the third parameter is consulted. If it is zero, the current process is suspended while the new process runs to completion including all its exit routines. A termination code and the child's exit code are placed where the sixth parameter points. The termination codes are

0 = normal *DosExit*, see exit code
1 = ended by hard I/O error
2 = ended by hardware trap
3 = ended by external signal (*DosKillProcess*)

The exit code is only significant when the termination code is zero; then it is the argument of the final *DosExit* call. For all other option values, the process ID of the new process is returned where the sixth parameter points. When the option is 1, 2 or 4 the process runs asynchronously. When the option is 4, it runs detached from any screen group; it cannot communicate with the screen, keyboard or mouse except in a Pop-up display, and all connection to the parent process is severed so *DosCWait* cannot be used to wait for its completion. When the option is 1 or 2, the new process is considered a child of the current process. *DosCWait* is used to wait for child processes to complete. If the option is 2, the child's termination and exit codes will be saved when it ends and can be retrieved

with *DosCWait*. When the option is 1, the codes are not retained and will be lost. When the option is 3, the new process is suspended for program stepping by *DosPTrace*. When the option is 5, it is created in a frozen state and can be thawed to run asynchronously with a *DosSystemService* call. The "object name buffer" specified in the first two parameters is not modified unless an error prevents creating a process. Then it receives a character string that helps explain the error, usually the filespec of the program file.

Discussion: *See* 6.1 **Include:** SESSION

Related: *DosCWait, DosPTrace, DosSystemService*

Family API: Only option zero (synchronous call) is permitted. The termination code will always be zero.

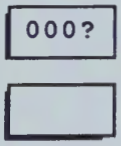
Errors to consider:

DosOpen error codes 2, 3, 4, 5, 32 and 123 may be returned

```
extern unsigned far pascal DOSEXECPGM(
                        char far *,     /* object buffer */
                        unsigned,       /* buffer size */
                        int,            /* option */
                        char far *,     /* argument */
                        char far *,     /* environment block or null */
                        PROCESSID far *,/* PID/termcode */
                        char far *);    /* filespec */
#define DosExecPgm DOSEXECPGM

function DosExecPgm (   vars AnObject : char;   { e.g. ObjStr[1] }
                             ObjSize : integer;
                             Option : integer;
                             ArgPtr : ADSMEM;   { use NilADS for nil }
                             EnvPtr : ADSMEM;   { use NilADS for nil }
                        vars TheAnswer : ProcessID; { TWO words! }
                      consts FileSpec : char    { e.g. Fspec[1] }
                        ) : word; external;
```

DosExit

End thread or whole process



If the first parameter is zero, the current thread is ended and the second parameter is ignored. When a thread is created with *DosCreateThread*, the address of a *DosExit(0,0)* call is pushed on its stack. This is not true of the first thread of a

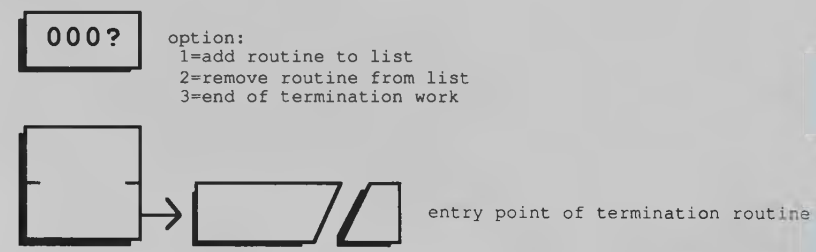
process; it must issue this call explicitly to end the process. If the first parameter is 1, the current process is terminated. If exit routines have been registered with *DosExitList*, the first thread of the process is diverted to execute them, while all other threads are suspended. Resources such as open files, allocated storage, and system semaphores are released. The second parameter becomes the result code returned to the parent process; if that was the command shell, the code can be tested in a batch command file as "errorlevel."

Discussion: *See* 9.1.4 **Include:** ASYNC
Family API: available
Errors to consider: none

```
extern void far pascal DOSEXIT(int, int);
#define DosExit DOSEXIT

procedure DosExit       (TheScope : integer; { zero for just me }
                         ExitCode : integer
                         ); external;
```

DosExitList

Add termination procedure



The first parameter specifies one of three operations related to the list of termination routines maintained for every process. If the parameter is 1, the routine addressed by the second parameter is added to the list. Its position in the list is unpredictable. If the parameter is 2, the routine addressed by the second parameter is removed from the list. When all threads of the current process have ended, the first thread is revived and set to work on the routines named in the exit list at that time. Each routine will be entered in its turn with SS and SP set to the top of the defined stack. The routine should do whatever is required to tidy up global resources that it knows about. It must *not* end by returning to its caller; it must repeat this call with a first parameter of 3. That signals the system to call the next routine in the exit list. When the option is 3, the second parameter is

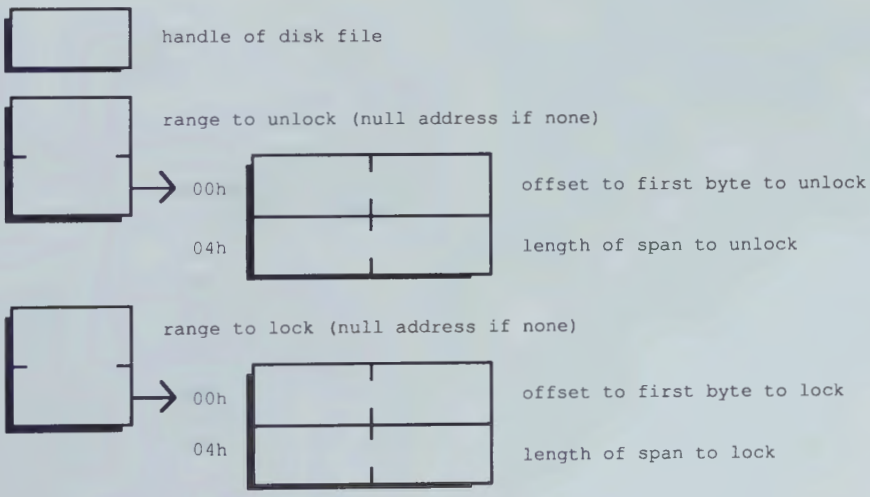
not inspected and may be any doubleword. Control never returns from a call with an option of 3.

Discussion: *See* 5.3.3 **Include:** PROCESS
Related: none
Family API: not available
Errors to consider:
13 A parameter is wrong: option is not 1, 2 or 3, or the routine has already been listed, or the routine is not on the list, or the address is invalid

```
extern unsigned far pascal DOSEXITLIST(int, ADSPROC );
#define DosExitList DOSEXITLIST

function DosExitList(   TheOption : integer;
                        TheCode : ADSPROC
                        ) : word; external;
```

DosFileLocks

Lock or unlock parts of a file



The file must be a disk file. Unlocking is done first. The offset and length of the unlocked range must be identical to a range previously locked. Locking is done next. If the range locked encompasses or overlaps on another locked range, whether locked by this process or another, an error code is returned. The offset and length may have any values; they don't need to fall in the bounds of the

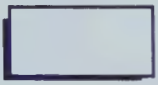
actual file size. Once a range has been locked, an attempt by a process other than this one to read, write, or lock bytes within the range causes an error return. Mutual exclusion is guaranteed; if multiple processes try to lock the same range, one will succeed and the others will see a lock violation. Access to locked ranges is kept on a per-process basis and is not inherited by a child process even if the file is inherited. Ranges that remain locked when the file is closed are unlocked in no predictable order.

Discussion: *See* 8.11.8 **Include:** FILES
Related: *DosOpen, DosRead, DosWrite*
Family API: only supported if "share" support is active
Errors to consider:
6 The handle must be open to a disk file
33 The unlock range doesn't match a previous lock range, or the lock range overlaps on a locked area of the file

```
/* use NULLRANGE as parameter to omit one or the other range */
extern unsigned far pascal DOSFILELOCKS(
                        HANDLE,
                        struct LockRange far *, /* unlock range */
                        struct LockRange far *, /* lock range */
                                );
#define DosFileLocks DOSFILELOCKS

function DosFileLocks  (TheHandle : Handle;
                       Unlock : ADSRange; { ADS MyRange or NullRange }
                        LockIt : ADSRange
                        ) : word; external;
```

DosFindClose

Release directory scan handle



handle returned by DosFindFirst

A directory scan begun with *DosFindFirst* is ended. The handle will no longer be recognized by *DosFindNext* or by *DosFindFirst* for reuse. Some small amount of system resource is released.

Discussion: *See* 8.6.5.2 **Include**: FILES

Related: *DosFindFirst, DosFindNext*

Family API: available

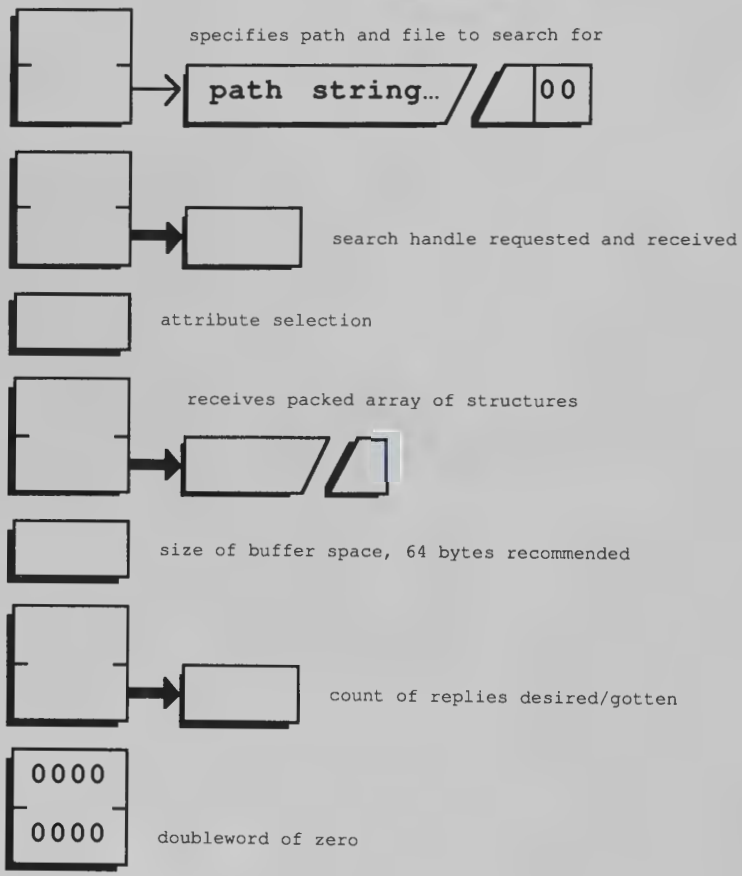
Errors to consider:

6 The handle must be one returned by *DosFindFirst* and not closed

```
extern unsigned far pascal DOSFINDCLOSE(HANDLE);
#define DosFindClose DOSFINDCLOSE

function DosFindClose  (TheHandle : Handle
                        ) : word; external;
```

DosFindFirst

Begin scan of directory



A scan is begun over one directory, looking for files whose names match a pattern. The path string specifies the drive, directory, and filename. The filename may include wildcard characters. The scan will always find all visible files, including those with the read-only and archived attributes. The attribute selection word can causes additional files to be returned if they exist:

- 0002h causes hidden files to be returned.
- 0004h causes system files to be returned.

- 0010h causes directories to be returned.

A data structure is returned for each matching file, up to the limit set by the size of buffer space and the input count word. See the discussion topic for the format of the data. The number of structures returned is set in the count word. A return code of zero means that *DosFindNext* may return more matching files. Error code 18 and count of zero means no files were found; code 18 and a nonzero count means all matching files were returned.

The output handle is for use with *DosFindNext*. It should finally be closed with *DosFindClose*. The handle word is also an input parameter. If it is -1 (0ffffh) it requests that a new handle be allocated to this scan. Otherwise it must be the handle of a prior, active scan, and requests that the prior scan be ended and the handle reused for this scan. A handle of 0001 is always available for reuse.

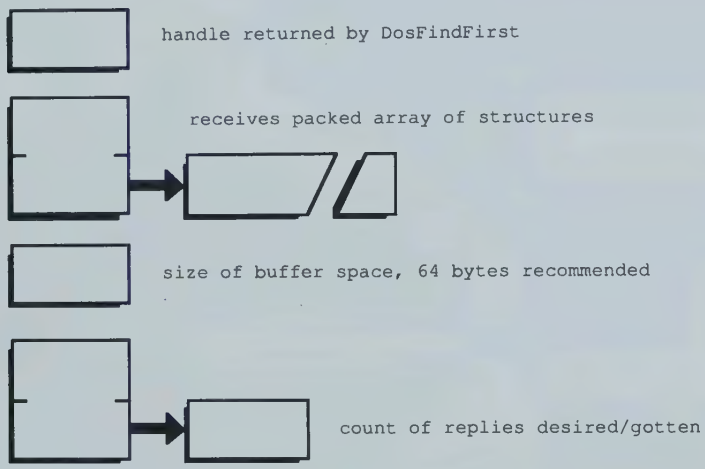
Discussion: *See* 8.6.5.2 through 8.6.5.5 **Include:** FILES
Related: *DosFindNext, DosSearchPath*
Family API: Input handle must always be 0001 (only one scan may be active)
Errors to consider:

3	Path not found: invalid drive or a member doesn't exist or isn't a directory
6	The input handle must be 1, -1, or an open handle returned by this call
18	No files matched (count=0) or all matching files were returned
87	The count must be nonzero and only the read-only, hidden, system, archive and directory attribute bits may be nonzero
111	The buffer size must be large enough for at least one entry

```
extern unsigned far pascal DOSFINDFIRST(
                        char far *,     /* path */
                        HANDLE far *,   /* request/return handle */
                        ATTRWORD,       /* requested attribute */
                        struct FileFind far *, /* buffer */
                        int,       /* buflen, eg sizeof(struct FileFind) */
                        int far *, /* request/return count */
                        long       /* zero */
                        );
#define DosFindFirst DOSFINDFIRST

function DosFindFirst  (consts ThePath : char;
                        vars   TheHandle : Handle;
                               TheAttr : AttrWord;
                        vars   Results : FileFind;
                               Buflen  : integer;
                        vars   TheCount : integer;
                               AZero : integer4
                        ) : word; external;
```

DosFindNext

Continue directory scan



A directory scan begun with *DosFindFirst* is continued. A data structure is returned for each matching file, up to the limit set by the size of the buffer and the input count word. The number of structures returned is set in the count word; it may be nonzero even when an error code is returned. An error code means that a subsequent call will return no files.

Discussion: *See* 8.6.5.2 **Include:** FILES
Related: *DosFindFirst, DosFindClose*
Family API: available
Errors to consider:

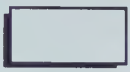
6 The handle must be one returned by *DosFindFirst* and not closed
18 No files matched (count=0) or all matching files were returned
87 The count must be nonzero
111 The buffer size must be large enough for at least one entry

```
extern unsigned far pascal DOSFINDNEXT(
                        HANDLE,
                        struct FileFind far *,
                        int,
                        int far *);
#define DosFindNext DOSFINDNEXT

function DosFindNext    (     TheHandle: Handle;
```

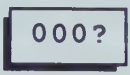
```
vars    Results: FileFind;
        Buflen : integer;
vars    TheCount: integer
) : word; external;
```

DosFlagProcess

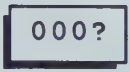
Send software signal to process



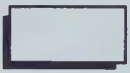
ID of process to signal




```
option: 1=signal only process
  0=signal process and all its children
```




```
flag number:
 0=flag A
 1=flag B
 2=flag C
```



word of information to accompany signal

If the specified process exists, it receives a signal event. If the second parameter is zero, a signal event will also be passed to any child processes created by that process (when option zero is used, the specified process must be a child of the current process). If the specified process has used *DosSetSigHandler* to block this flag type, an error code is returned. If a signal of this flag type is already being held for the specified process, an error code is returned. If a process has not set up a signal handler (with *DosSetSigHandler*) the signal will be received by the default signal handler, whose action is to acknowledge it but ignore it. No indication of this is returned, since the signal has been delivered even though the process will make no response. If a process has used *DosHoldSignal* to suspend signalling, the signal will be held for that process. No indication of this is returned; the system assumes that the signal will finally be delivered. The signal is delivered when the handler for the flag type is invoked. Eventually the signalled process should use *DosSetSigHandler* to acknowledge the signal. Until it does so, signals of this type are considered to be held. A second signal before acknowledgement is held; a third is rejected.

Discussion: *See* 5.3.1 **Include:** PROCESS
Related: *DosSetSigHandler, DosHoldSignal, DosSendSignal*
Family API: not available
Errors to consider:

1 The second parameter must be zero or 1
156 The specified process has blocked these signals
162 A signal of this type is already pending at that process
186 The third parameter must be zero, 1 or 2
303 The first parameter must be an existing process
305 When the option is zero, the process must be a child of this one

```
extern unsigned far pascal DOSFLAGPROCESS(
                                PROCESSID,
                                int,            /* 0 or 1 */
                                int,            /* flag 0/1/2 */
                                unsigned);      /* argument */
#define DosFlagProcess DOSFLAGPROCESS

function DosFlagProcess(TheProc : ProcessID;
                        Option : integer;       { 0 or 1 }
                        FlagNo : integer;       { 0/1/2 }
                        TheInfo : integer       { argument }
                        ) : word; external;
```

DosFreeModule

Release link to dynamic link module

handle of dynamic link module

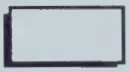
The first parameter must be a handle for a module bound to the current process. Usually the handle will be one returned by *DosLoadModule* or *DosGetModHandle*, but it may be the one passed into a module when it is initialized, and saved since then. The binding between the process and the dynamic link module is broken; that is, the segments of the module are deleted from the LDT of the process so they are no longer addressable. If the module was bound at load time, references to its segments may still exist in the machine code of the process, but an attempt to address the module's segments will cause a segmentation error. The module's use count is decremented; if it goes to zero, the module is unloaded from storage.

Discussion: *See* 4.5 **Include:** PROCESS
Related: *DosLoadModule*
Family API: not available
Errors to consider:
6 The handle must that of a module bound to the current process

```
extern unsigned far pascal DOSFREEMODULE(HANDLE);
#define DosFreeModule DOSFREEMODULE

function DosFreeModule( TheHandle:Handle
                        ) : word; external;
```

DosFreeSeg

Release storage segment



selector for segment

If the selector names a segment that is part of the address space of the current process, that segment is removed from the address space. Its descriptor is made invalid in the process's Local Descriptor Table; an attempt to use the selector in addressing will terminate the program with a protection error. Any segment may be released, regardless of how it was obtained (even the program's active code, data or stack segments can be released, though this produces immediate termination for a protection error). Unshared segments are returned to the system's pool of vacant storage. Releasing a shared segment makes it unavailable to the current process and decrements its reference count; if that goes to zero the segment will be released. Only the first selector of a huge segment is acceptable to *DosFreeSeg*. Releasing the first segment of a Huge allocation releases all the allocation and invalidates all its descriptors.

Discussion: *See* 7.3.8 **Include:** STORAGE

Related: *DosAllocHuge, DosAllocSeg, DosAllocShrSeg, DosGetShrSeg, DosGetSeg, DosGiveSeg*

Family API: Storage is not made physically inaccessible.

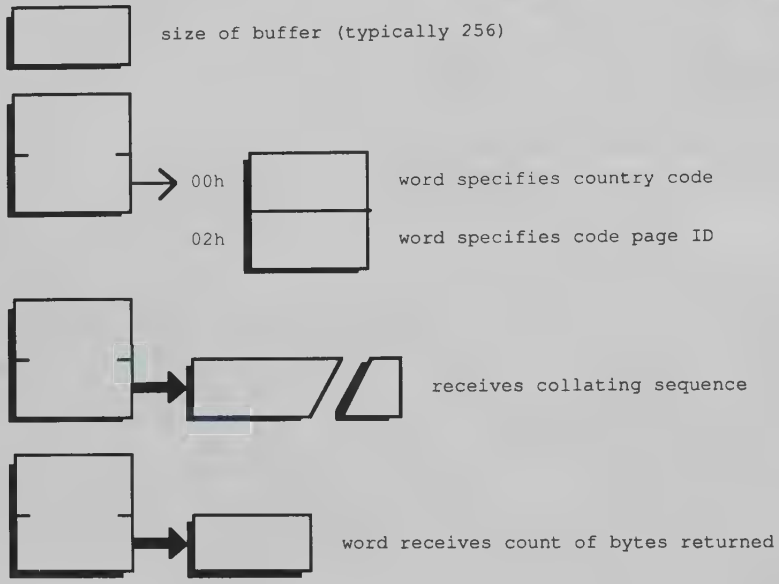
Errors to consider:

5 That segment has already been freed or is a middle part of a huge allocation

```
extern unsigned far pascal DOSFREESEG( SELECTOR );
#define DosFreeSeg DOSFREESEG

function DosFreeSeg     (TheSelector:Selector
                         ) : word; external;
```

DosGetCollate

Get national collating sequence for ASCII



The first word of the second parameter specifies a country number (see Table 5.2, p. 160 or zero to indicate the country set in the configuration file. The second word specifies a code page (see Table 5.3, p. 162), or zero to indicate the primary code page for the specified country. A table of up to 256 bytes is copied into the buffer specified by the first and third parameters, and the amount copied is set in the fourth parameter. Each table byte gives the collating value of the corresponding ASCII byte. That is, for ASCII bytes X and Y, if TABLE[X] is less than TABLE[Y], byte X should collate (sort) ahead of byte Y.

Discussion: *See* 5.2.2 **Include:** SYSINFO

Related: *DosCaseMap*, *DosGetCP*, *DosGetCtryInfo*

Family API: available

Errors to consider:

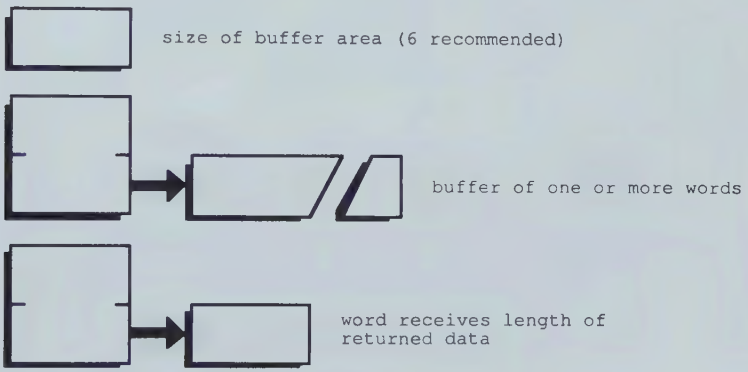
396 Couldn't locate the COUNTRY.SYS file
397 Couldn't open the COUNTRY.SYS file
398 The country number must be one supported by OS/2
399 The buffer wasn't large enough for the entire data structure

```
extern unsigned far pascal DOSGETCOLLATE(
                        int,            /* size of buffer */
```

```
                        struct CTRYCPID far *,
                        char far *,     /* buffer */
                        int far *);     /* returned length */
#define DosGetCollate DOSGETCOLLATE

function DosGetCollate (     BuffSize : integer;
                        vars TheCtry : CtryCPID;
                        vars Buffer  : char; { e.g. ColBuf[1] }
                        vars SizeUsed : integer
                        ) : word; external;
```

DosGetCP

Query current code page



The numbers of the logical code pages that were loaded wen the system started up (in response to *codepage=* and *country=* configuration statements) are returned as a series of binary integers in the buffer described by the first and second parameters. The amount of data copied is returned in the third parameter.

Discussion: *See* 5.2.3 **Include:** SYSINFO

Related: *DosGetCollate, DosGetCtryInfo, KbdGetCP, VioGetCP*

Family API: available

Errors to consider:

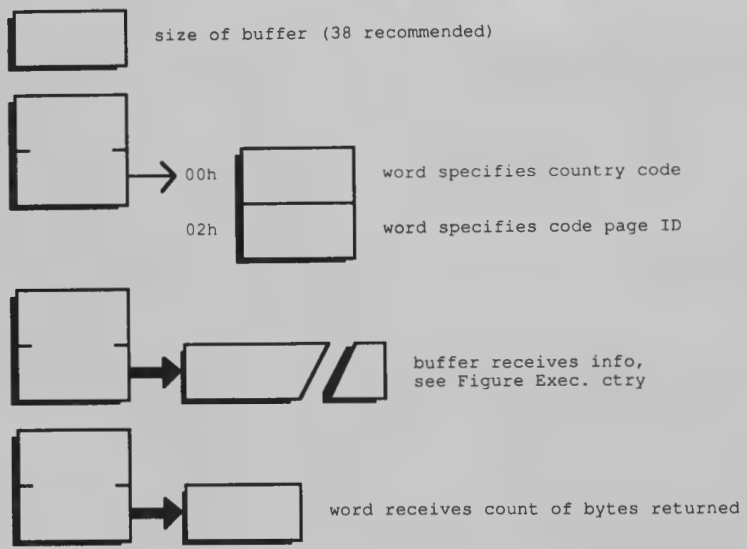
399 The data that to be returned was longer than the buffer and was truncated

```
extern unsigned far pascal DOSGETCP(
                        int,
                        int far *,
                        int far *);
#define DosGetCP DOSGETCP

function DosGetCP (          BuffSize : integer;
```

```
        Buffer : ADSMEM; { e.g. ADS CtryID }
    vars RetSize : integer
    ) : word; external;
```

DosGetCtryInfo

Query national data formats



The first word of the second parameter specifies a country code (see Table 5.2 on p. 160) or contains zero to indicate the country set at configuration time. The COUNTRY.SYS file is indexed to find data for the specified country. The data structure shown in Figure 5.7 on p. 159, or as much of it as will fit in the size given in the first parameter, is copied into the specified buffer and the length copied is set in the last parameter.

Discussion: *See* 5.2.1 **Include:** SYSINFO
Related: *DosGetCollate*, *DosGetCP*
Family API: available
Errors to consider:
396 Couldn't locate the COUNTRY.SYS file
397 Couldn't open the COUNTRY.SYS file
398 The country number must be one supported by OS/2
399 The buffer wasn't large enough for the entire data structure

```
extern unsigned far pascal DOSGETCTRYINFO(
                        int,
                        struct CTRYCPID far *,
                        struct CTRYINFO far *,
                        int far *);
#define DosGetCtryInfo DOSGETCTRYINFO

function DosGetCtryInfo (    BuffSize : integer;
                        vars TheCtry : CtryCPID;
                        vars TheInfo : CtryInfo;
                        vars RetSize : integer
                        ) : word; external;
```

DosGetDateTime

Query date and time information

date and time structure, see Figure Exec.Date

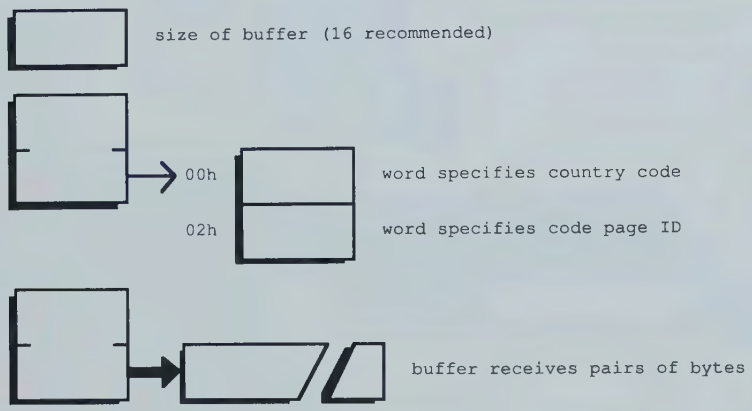
The time (hour, minute, second, hundredths of second, and time zone) items, and the date (day, month, year and day of week) items are placed in the structure. The same structure may be input to *DosSetDateTime*. The same information is available in slightly different order as part of the global information segment returned by *DosGetInfoSeg*.

Discussion: *See* 5.1.2 **Include:** SYSINFO
Related: *DosSetDateTime*
Family API: available
Errors to consider: none

```
extern void far pascal DOSGETDATETIME(struct DATETIME far *);
#define DosGetDateTime DOSGETDATETIME

Procedure DosGetDateTime(vars TheInfo: DateTime);
                        external;
```

DosGetDBCSEv

Query double-byte character set status



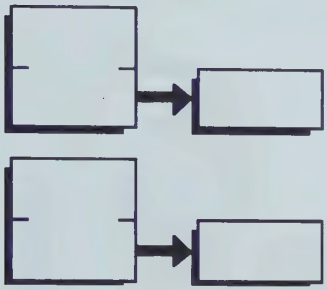
The first word of the second parameter specifies a country number (see Table 5.2, p. 160 or zero to indicate the country set in the configuration file. The second word specifies a code page (see Table 5.3, p. 162), or zero to indicate the primary code page for the specified country. A series of pairs of bytes (usually only one pair and probably not more than five pairs) is returned in the buffer described by the first and third parameters. Each pair of bytes represents a range of byte values that, in the specified code page, are lead-in bytes to double-byte characters. The final (or only) pair of bytes is a pair of bytes with binary value zero to mark the end of the table. If the entire table won't fit in the buffer given, as much as will fit is copied and an error code is returned.

Discussion: *See* 5.2.4 **Include:** SYSINFO
Related: *DosGetCP, DosGetCollate, DosSetCP*
Family API: available
Errors to consider:
399 There was more data to return than would fit in the buffer

```
extern unsigned far pascal DOSGETDBCSEV(
                        int,
                        struct CTRYCPID far *,
                        char far *);
#define DosGetDBCSEv DOSGETDBCSEV

function DosGetDBCSEv (       BuffSize : integer;
                        vars  TheCtry : CtryCPID;
                        vars  Buffer : char
                        ) : word; external;
```

DosGetEnv

Get selector for environment segment



word receives selector of environment segment

word receives offset of parameter strings

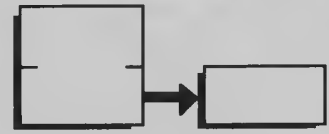
The selector for the environment segment associated with the current process is returned in the first parameter. The same selector is passed in register AX when a process is started. The environment is a collection of ASCII strings, each of which is terminated by a null character (byte of 00h). At offset zero in the segment is a pool of environment variable strings such as is maintained with the PATH, DPATH, and SET commands. A null string marks the end of the pool of variables. Beyond the variable-strings in the segment is another set of strings that were prepared by the parent of this process. If the parent was the command shell, the first string will be the command verb (the first word from the command line, usually the filename of the program) and the second will be the command tail, that is, the remainder of the command line exactly as typed (which may be the null string). The offset to these parameter strings is returned in the second parameter. A null string marks the end of the parameter strings.

Discussion: *See* 5.1.5 **Include:** SYSINFO
Related: *DosExecPgm*
Family API: available
Errors to consider: none

```
extern unsigned far pascal DOSGETENV(
                                    unsigned far *,  /* selector */
                                    unsigned far *); /* offset */
#define DosGetEnv DOSGETENV

procedure DosGetEnv     (vars TheSel: word; { s.b. an ADS type .S field }
                         vars ParmOff: word { s.b. an ADS type .R field }
                        ); external;
```

DosGetHugeShift

Get inter-selector constant



receives shift value

The Huge Shift is a constant that is used to derive the selector numbers for successive segments of a "huge" storage block. The difference between one selector and the next is 0001h shifted left by the Huge Shift. The Huge Shift constant is also available as a field of the system information structure returned by *DosGetInfoSeg*.

Discussion: *See* 7.3.3.1 **Include:** STORAGE

Related: *DosAllocHuge*

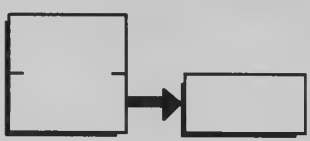
Family API: Shift is always 12, yielding a difference of 1000h, the difference between physical segment addresses of adjacent segments.

Errors to consider: none

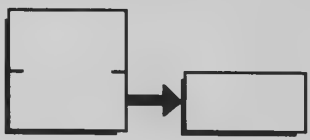
```
extern unsigned far pascal DOSGETHUGESHIFT(int far *);
#define DosGetHugeShift DOSGETHUGESHIFT

function DosGetHugeShift(vars TheShift:integer
                         ) : word; external;
```

DosGetInfoSeg

Get selectors of information segments



word receives selector of global segment



word receives offset of local segment

The system maintains a global segment packed with useful information that it

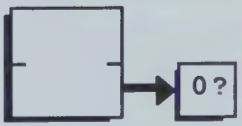
updates continuously. A selector for this segment, described as a read-only data segment, is returned in the first parameter word. Install this word as the more significant word of a far pointer with an offset of zero in order to address the segment. The layout of the segment is shown in Figure 5.3, p. 148. The date and time information from the global information segment is available in a different arrangement via the *DosGetDateTime* call. The system maintains a smaller segment of per-process information for each process. This call returns a selector for that segment, also described as a read-only data segment. The layout of the local segment is shown in Figure 5.5, p. 153. Some of the information from the local information segment is available in a different arrangement via the *DosGetPID* call.

Discussion: *See* 5.1.4 **Include:** SYSINFO
Related: *DosGetDateTime, DosGetPID*
Family API: not available
Errors to consider: none

```
extern void far pascal DOSGETINFOSEG(
                                SELECTOR far *,  /* global */
                                SELECTOR far *); /* local */
#define DosGetInfoSeg DOSGETINFOSEG

procedure DosGetInfoSeg (vars GlobSel, LocSel : Selector );
                        external;
```

DosGetMachineMode

Query CPU mode



byte receives 01 if protect mode

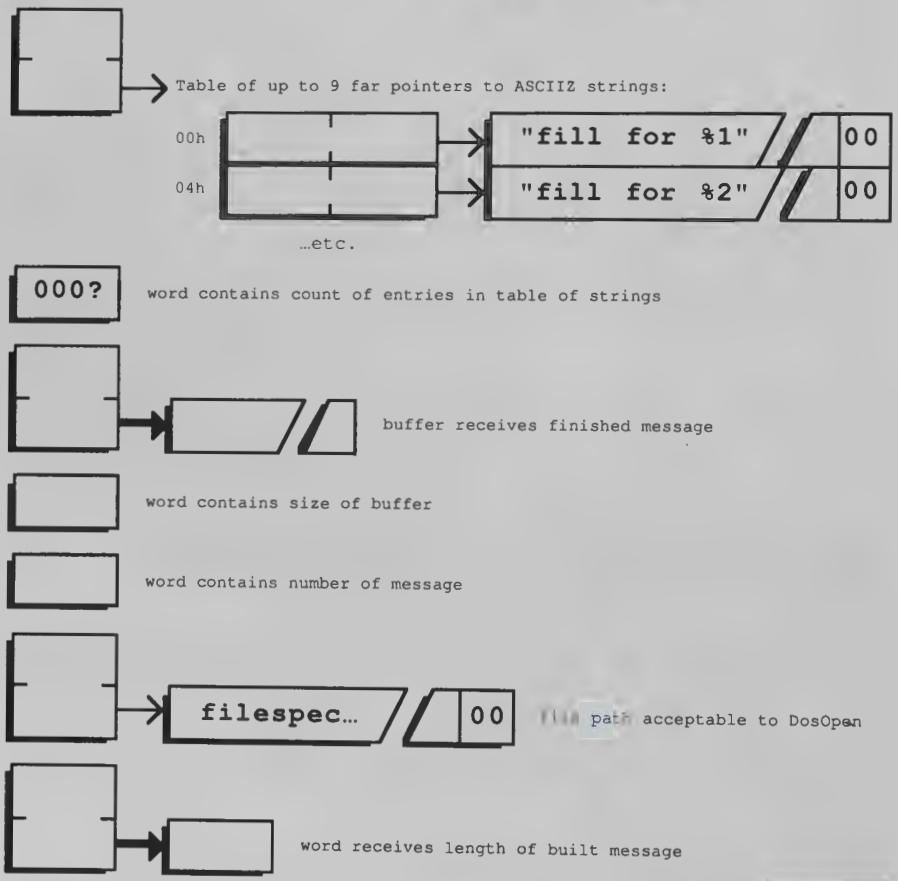
If the CPU is operating in Protect mode, 01h is deposited in the specified byte. Zero is put there in Real mode (by the Family API library routine).

Discussion: *See* 5.1.1.1 **Include:** SYSINFO
Related: *DosDevConfig*
Family API: available
Errors to consider: none

```
extern void far pascal DOSGETMACHINEMODE(char far *);
#define DosGetMachineMode DOSGETMACHINEMODE

procedure DosGetMachineMode(vars TheMode:byte);
                        external;
```

DosGetMessage

Prepare message from message file



A message template is located and, if it contains marked insertion points, they are filled in from a table of insertion strings. The message is selected by the number in the fifth parameter and the filename in the path given as the sixth parameter. This file must be the output of the MKMSGF utility. The service looks first to see if the specified message has been extracted from the file and bound into the program image by the MSGBIND utility; if so, the message template in storage

is used. Otherwise the specified file is opened. If the sixth parameter includes a path only that path is searched; otherwise the file will be found if it is in the current directory, the root directory of the boot drive, or along the DPATH path list. If the file cannot be found or if the message cannot be found in the file, an error code will be returned; however, a default message template (a "DOSnnnn" error message) will be used. Once the file has been opened, it remains open until the program ends or explicitly closes it.

When the message template is found it is copied into the buffer whose address and size are given in the third and fourth parameters. In copying the system scans for insertion markers of the form "*%n*" where *n* is a decimal digit from 1 up to the number insertion strings specified by the second parameter. When a marker is found, it is replaced with the corresponding string from the table of strings specified in the first parameter. Markers whose digits exceed the table size, and percent signs followed by other characters, are copied without change. If in copying the template or an insertion string, the output reaches the size of the buffer, copying stops and an error code is returned. (This can also happen when a default error message is used, in which case the "out of room" error supersedes the original error code.) The number of bytes copied to the buffer is returned in the last parameter. No final null byte is copied; a null must be appended if one is required (the insertion strings and filespec string are ASCIIZ strings, but the template and output message are not). The message template usually ends in CR and LF bytes left by MKMSGF.

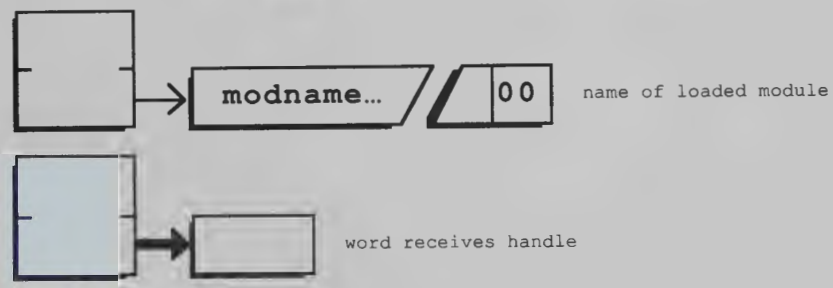
Discussion: *See* 5.4 **Include:** MESSAGE
Related: *DosInsMessage, DosPutMessage*
Family API: not available
Errors to consider:
316 The message text had to be truncated to fit the buffer
317 The message number must be defined in the message file
318 The message file could not be opened (might also be indicated by DosOpen error codes 2, 3, 4, 5, 32 or 123)
319 The message file named must be the output of MKMSGF
320 The size of the insertion table must lie between zero and 9

```
extern unsigned far pascal DOSGETMESSAGE(
                char far * far *,       /* table of string ptrs */
                int,                    /* number of string ptrs */
                char far *,             /* return buffer */
                int,                    /* max return size */
                int,                    /* message number */
                char far *,             /* message file spec */
                int far *);             /* output length */
#define DosGetMessage DOSGETMESSAGE

function DosGetMessage( consts Inserts  : MsgINSList;
```

```
        InsCount : integer;
vars    Buffer   : char; { char [1] of buffer }
        BuffSize : integer;
        MsgNumber: integer;
consts MsgFile  : char; { char [1] of asciiz }
vars    MsgLength: integer
) : word; external;
```

DosGetModHandle

Get handle for loaded module



The ASCIIZ name string specified is looked up in the current process's table of bound modules. If it does not match the name of a module that has been loaded, an error code is returned. The match ignores letter case; the names compared are the module names defined by the LIBRARY statement in a link definition file, which are not necessarily the same as the executable filenames. If the name is matched, a handle for the module is returned which may be used in *DosFreeModule*, *DosGetModName*, *DosGetProcAddr*, and *DosGetResource*.

Discussion: *See* 4.5 **Include:** PROCESS

Related: *DosLoadModule*, *DosFreeModule*, *DosGetModName*, *DosGetProcAddr*, *DosGetResource*

Family API: not available

Errors to consider:

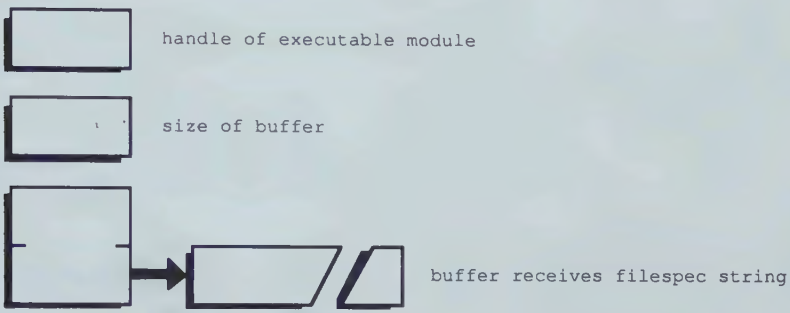
126 The module name must match the name of a package now in storage.

```
extern unsigned far pascal DOSGETMODHANDLE(
                                char far *,
                                HANDLE far *);
#define DosGetModHandle DOSGETMODHANDLE

function DosGetModHandle( consts ModName: char; { e.g. NameStr[:] }
                          vars TheHdle: Handle
```

```
) : word; external;
```

DosGetModName

Get filespec of loaded module



The first parameter must be a handle for a module that has been loaded into storage (usually a handle returned by *DosLoadModule* or *DosGetModHandle*, but it may be one passed into a module when it was initialized and saved since then). The fully-qualified filespec (drive, path, filename and extension) of the executable file for the module is returned in the buffer. The drive and path will be those given in the *libpath*= configuration statement and the file extension will be .DLL or .EXE. The buffer should be at least 128 bytes in size, although the filespec will usually be shorter.

Discussion: *See* 4.5 **Include:** PROCESS

Related: *DosGetModHandle, DosLoadModule*

Family API: not available

Errors to consider:

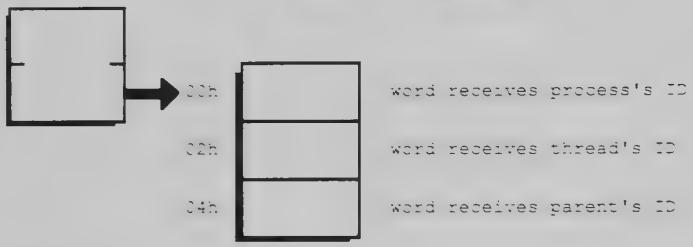
6 The handle must specify a module now in storage

24 The filespec string did not fit the buffer, try a larger buffer

```
extern unsigned far pascal DOSGETMODNAME(
                                HANDLE,
                                int,
                                char far *);
#define DosGetModName DOSGETMODNAME

function DosGetModName(      TheHandle : Handle;
                             BuffSize : integer;
                         vars Buffer: char { e.g. MyBuff[1] }
                         ) : word; external;
```

DosGetPID

Query process and thread IDs

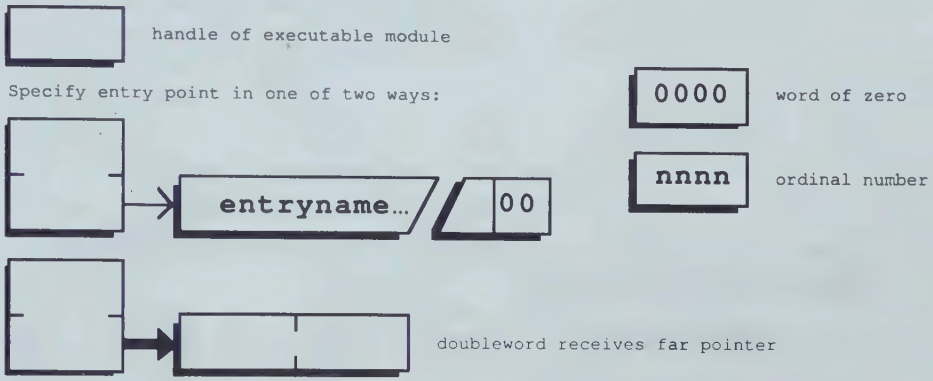


The Process ID of the running process and of its parent are returned. The thread ID of the running thread is also returned. The same information is available in a slightly different arrangement in the local information segment returned by *DosGetInfoSeg*.

Discussion: *See* 5.1.4.3.1 **Include:** SYSINFO
Related: *DosGetInfoSeg*
Family API: available
Errors to consider: none

```
extern void far pascal DOSGETPID( struct PROCIDSET far * );
#define DosGetPID DOSGETPID

procedure DosGetPID     (vars TheStuff: ProcIDSet);
                         external;
```

DosGetProcAddr



The first parameter must be a handle for a module bound to the current process. Usually the handle will be one returned by *DosLoadModule* or *DosGetModHandle*, but it may be one given to a module when it was initialized, and saved since then.

The second parameter specifies an entry point to the module that appears in that module's table of entry points. This may be a far pointer to an ASCIIZ string. When it is, and if the first character of the string is *not* the number-sign "#," the string is taken as the public name of an entry point as defined in an EXPORTS statement when the module was linked. If the second parameter is a string that begins with the character "#," the rest of the string is taken as a decimal number, the ordinal number of the entry point as defined in an EXPORTS statement when it was linked. Alternatively, the selector portion of the second parameter may be zero (making it invalid as an address), in which case the offset word is taken instead as an entry ordinal number in binary. The far address of the designated entry point is returned in the third parameter, which may be used to access the entry point.

Discussion: *See* 4.5 **Include:** PROCESS

Related: *DosFreeModule, DosGetModHandle, DosGetResource, DosLoadModule*

Family API: not available

Errors to consider:

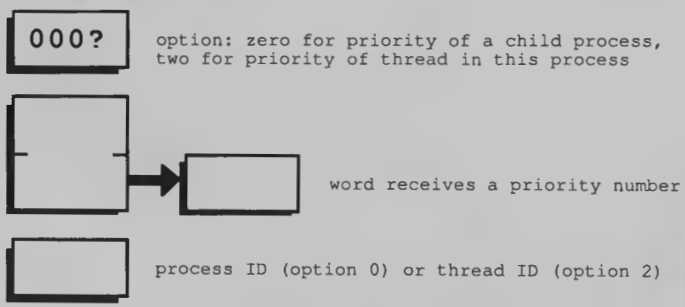
6 The handle must specify a module bound to the current process

127 The entry point must be one specified in the Linker EXPORTS list

```
extern unsigned far pascal DOSGETPROCADDR(
                                HANDLE,
                                char far *,
                                union ADS far *);
#define DosGetProcAddr DOSGETPROCADDR

function DosGetProcAddr(      TheHdle : Handle;
                              TheEntry : ADSMEM;
                        vars  TheAddr : ADSPROC
                        ) : word; external;
```

DosGetPrty

Query thread priority



The priority of some thread is returned. If the first parameter is zero, the third parameter must contain either the process ID number of an existing process or zero to indicate the current process. The priority of the first thread of the specified process is returned. (Only the first thread of another process may be queried. In the event that the process exists but its first thread has terminated, an invalid thread ID is indicated.) If the first parameter is 2, the third parameter may contain either the thread ID of a thread in the current process or zero to indicate the current thread. The priority of the indicated thread is returned. The form of the returned word is 0CPPh, where the most significant byte has the dispatching class, 01 for idle-time, 02 for regular, and 03 for time-critical. The least significant byte contains the priority and will vary from 00h (the usual value) up to 1fh or 31 decimal.

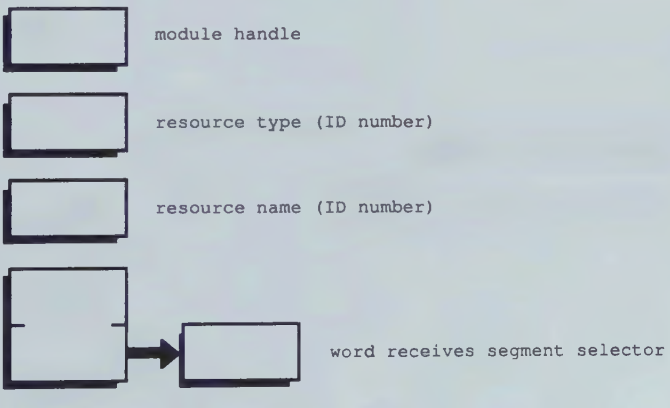
Discussion: *See* 9.1.5 **Include:** ASYNC
Related: *DosCreateThread, DosSetPrty*

Family API: not available
Errors to consider:
303 The process specified must exist
308 The first parameter must be zero or 2.
309 A thread ID must identify an existing thread of the current process (if option=0, the first thread of the process has ended)

```
extern unsigned far pascal DOSGETPRTY(
                        int,            /* 0 or 2 */
                        int far *,      /* receives prty */
                        THREADID);      /* or PROCESSID */
#define DosGetPrty DOSGETPRTY

function DosGetPrty     ( TheOption : integer;
                          vars ThePrty : integer;
                          TheObject : ThreadID
                        ) : word; external;
```

DosGetResource

Get selector for resource



The first parameter must be a handle for a module bound to the current process. Usually the handle will be one returned by *DosLoadModule* or *DosGetModHandle*, but it may be the one passed into a module when it is initialized, and saved since then. (For this call the handle might well be the process's own handle, passed in register DI when the process was started.)

A resource is a read-only segment of data that may attached to the executable file

of a program. Resources are primarily used by programs running under the OS/2 Presentation Manager. Such programs attach a variety of binary images (fonts, cursor shapes, etc.) as resources. Each resource is identified by its 16-bit type number, and within its type by a "name" which is also a 16-bit number. The first 256 type numbers are reserved by the Presentation Manager, and the first ten have predefined meanings.

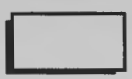
If the specified resource is part of the specified module, it is set up in a read-only storage segment, and the segment selector is returned. The program is expected to know the size of the resource.

Discussion: *See* 4.5.2 **Include:** PROCESS
Related: *DosLoadModule, DosGetModHandle*
Family API: not available
Errors to consider: none

```
extern unsigned far pascal DOSGETRESOURCE(
                                HANDLE,
                                unsigned, unsigned,
                                SELECTOR far *);
#define DosGetResource DOSGETRESOURCE

function DosGetResource(     ModHdle : Handle;
                             TheType, TheName : word;
                         vars TheSelector : Selector
                         ) : word; external;
```

DosGetSeg

Gain access to shared segment



selector of segment

A segment of storage must have been allocated with the *DosAllocSeg* option that made it elegible for sharing with this call. The selector that was returned to the allocating process is the input parameter to this call. This call is executed by the process that wants to share the storage, not by the process that allocated it. The segment is added to the address space of the current process with the identical selector. Since the selector is the same in all processes that have access, a far pointer to the contents of the segment is relevant in any process. The system keeps a count of the processes that have access to the segment, and retains the segment until none do. Despite the similarity of their names, there is no

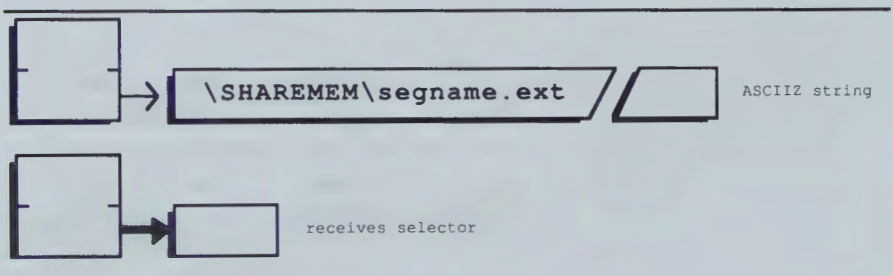
relationship between this call and *DosGiveSeg*.

Discussion: *See* STORE.SHARE.GET **Include:** STORAGE
Related: *DosAllocSeg, DosFreeSeg*
Family API: not available
Errors to consider:
5 The requested segment doesn't exist or wasn't allocated with the right option

```
extern unsigned far pascal DOSGETSEG( SELECTOR );
#define DosGetSeg DOSGETSEG

function DosGetSeg      ( TheSelector : Selector
                        ) : word; external;
```

DosGetShrSeg

Get access to named shared segment



If the segment with the given name exists, it is added to the address space of the current process. A selector for it is returned. Any number of processes may have access to the segment, and all may modify its contents. The selector for the segment is not necessarily the same in each process, so a far pointer into the segment is not relevant in another process. The system keeps a count of the number of processes that have access to the segment. When the last one frees the segment or terminates, the segment is released.

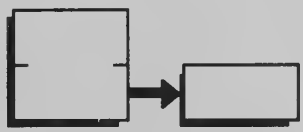
Discussion: *See* 7.4.1 **Include:** STORAGE
Related: *DosAllocShrSeg, DosFreeSeg*
Family API: not available
Errors to consider:
2 No segment of that name can be found
123 The segment name must be an ASCIIZ string of the form "\SHAREMEM\segname.ext" and use only characters valid in filenames

```
extern unsigned far pascal DOSGETSHRSEG(
                        char far *,
                        SELECTOR far *);
```

```
#define DosGetShrSeg DOSGETSHRSEG

function DosGetShrSeg  (vars TheName: char; { e.g. AString[1] }
                        vars TheSelector: Selector
                        ) : word; external;
```

DosGetVersion

Get OS/2 version numbers



two bytes receive minor and major numbers

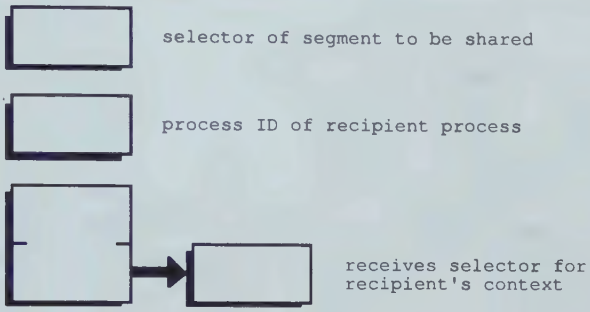
The OS/2 version is stored. It may be treated as a 16-bit word or as two bytes. Considered as a word, the decimal value of the first release of OS/2 is 2,560 (0a00h). The version-word of a subsequent version will be a larger number. As two bytes, the least significant (at the lower address) receives a minor version number that reflects maintenance. The most significant byte (at the higher address) receives the major number that reflects major functional releases. The first release of OS/2 has minor version zero, major version 10.

Discussion: *See* 5.1.1.2 **Include:** SYSINFO
Related: none
Family API: available
Errors to consider: none

```
extern void far pascal DOSGETVERSION(unsigned far *);
#define DosGetVersion DOSGETVERSION

procedure DosGetVersion (vars VerWord: word);
                        external;
```

DosGiveSeg

Give access to shared segment



A segment of storage must have been allocated (via *DosAllocSeg* or *DosAllocHuge*) using the option making it elegible for sharing via this call. This call is executed by the same process that performed the allocation. The selector returned by allocation is the first parameter. The second is the process ID number of the process that is to share the storage, usually a child process. The segment is added to the address space of the recipient process at the first available selector. That selector is returned. It is only relevant in the context of the recipient address space, and must somehow be conveyed to the recipient process. The system keeps a count of the processes that have access to the segment, and retains the segment until none do. Despite the similarity of their names, there is no relationship between this call and *DosGetSeg*.

Discussion: *See* 7.4.3 **Include:** STORAGE

Related: *DosAllocSeg, DosAllocHuge, DosFreeSeg*

Family API: not available

Errors to consider:

5 The specified segment doesn't exist or wasn't allocated with the proper option, or the specified process doesn't exist

```
extern unsigned far pascal DOSGIVESEG(
                        SELECTOR,
                        unsigned,
                        SELECTOR far *);
#define DosGiveSeg DOSGIVESEG

function DosGiveSeg     (     MySel: Selector;
                              YourPID: word;
                        vars YourSel: Selector
                        ) : word; external;
```

DosHoldSignal

Block or reenable external signals

```
action:
 0=enable signals
 1=hold them
```

Receipt of the six external signals (Control-C, Control-Break, process termination, and Flags A, B and C) is controlled by a count maintained for each process. If the count is nonzero, signals remain pending but do not take effect. If the parameter to this call is 1, the count for the current process is incremented, thus blocking receipt of signals. If the parameter is zero and the count is nonzero, the count is decremented. This may or may not enable receipt of signals, depending on the state of the count. When the count is decremented from one to zero, any pending signals are immediately received.

Discussion: *See* 5.3.1, 5.3.1.2 **Include:** PROCESS

Related: *DosSetSigHandler*

Family API: Only the Control-C signal is blocked by this mechanism

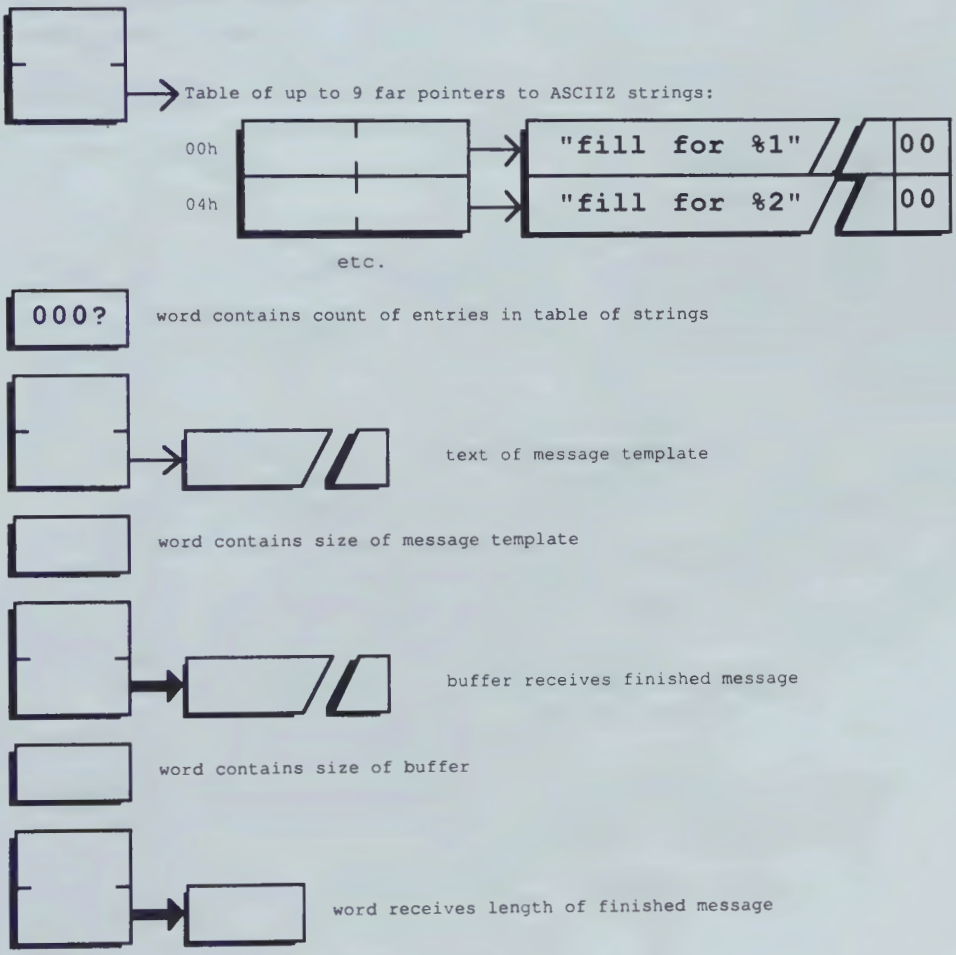
Errors to consider:

1 The action must be 0 or 1, and if 0 the count must be nonzero

```
extern unsigned far pascal DOSHOLDSIGNAL(int);
#define DosHoldSignal DOSHOLDSIGNAL

function DosHoldSignal (Option : integer
                        ) : word; external;
```

DosInsMessage

Prepare message template with insertions



The text of the message template specified by the third and fourth parameters is copied into the buffer specified by the fifth and sixth parameters. As it copies text, the system inspects for insertion points marked by "*%n*" where *n* is a decimal digit from 1 to up to the count of entries given in the second parameter. The insertion string from the table is copied in place of the marker. Markers in which *n* exceeds the size of the table, and percent signs not followed by digits, are

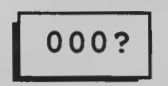
copied without change. If the copied text reaches the size of the buffer, copying stops and an error code is returned; otherwise the full template is copied. The number of bytes copied is returned in the word given as the last parameter. The insertion strings are null-delimited, but no null terminator is added to the output string. If the output is to be terminated by a null or a CR-LF pair, these characters must either be part of the template (and counted in its length) or else must be added afterward.

Discussion: *See* 5.4 **Include:** MESSAGE
Related: *DosGetMessage, DosPutMessage*
Family API: available
Errors to consider:
316 The message text had to be truncated to fit the buffer
320 The size of the insertion table must lie between zero and 9

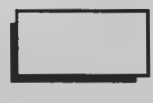
```
extern unsigned far pascal DOSINSMESSAGE(
                char far * far *,       /* table of string ptrs */
                int,                    /* number of string ptrs */
                char far *,             /* input text */
                int,                    /* input length */
                char far *,             /* return buffer */
                int,                    /* max return size */
                int far *);             /* output length */
#define DosInsMessage DOSINSMESSAGE

function DosInsMessage( consts Inserts  : MsgINSList;
                              InsCount  : integer;
                        consts Template : char; { char [1] of text }
                              TempSize  : integer;
                        vars   Buffer   : char; { char [1] of buffer }
                              BuffSize  : integer;
                        vars   MsgLength : integer
                        ) : word; external;
```

DosKillProcess

Terminate a process




```
option:
 0=terminate only the specified process
 1=terminate it and all its children
```



process ID to be terminated

When the option is 1 (kill specific process) the specified process is looked up. If it exists anywhere in the system, and if it has not used *DosSetSigHandler* to ignore, block or handle the termination signal, it is terminated: it ceases normal execution; if it has set up exit routines with *DosExitList*, they are invoked; then it ends. When the option is zero (kill entire tree of processes) the specified process must apparently be a child of the current process. Then it and all of its descendants are terminated. If a terminated process has set up a handler for the termination signal, the handler receives control and it may choose to end the process or not, as it pleases. If it has blocked the termination signal, an error code is returned.

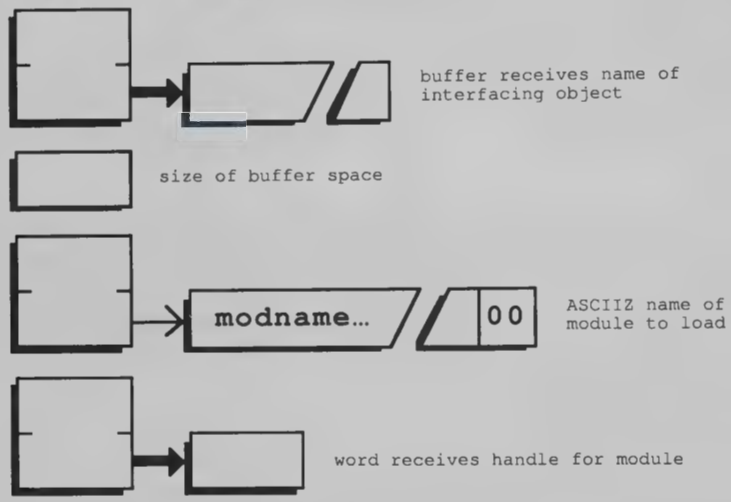
This powerful service ends any process that has not acted to deflect the termination signal. It can terminate the calling process, or a command shell (including the one that is the ancestral parent of the calling process). If a sufficiently important process is terminated, the system may become unusable.

Discussion: *See* 5.3.1 **Include:** PROCESS
Related: *DosSetSigHandler*
Family API: not available
Errors to consider:
13 The first parameter must be zero or 1
156 That process has blocked the termination signal
162 A termination signal is already being held for that process
303 The second parameter must be an existing process
305 Option is zero but target process is not a child of this process

```
extern unsigned far pascal DOSKILLPROCESS(
                                PROCESSID,
                                int);    /* option 0 or 1 */
#define DosKillProcess DOSKILLPROCESS

function DosKillProcess(TheProcess : ProcessID;
                        Option : integer { option 0 or 1 }
                        ) : word; external;
```

DosLoadModule

Bind dynamic link package to process



The third parameter must be a simple name of 1-8 characters (a filename only, no drive, path, or file extension). This name is looked up in the OS/2 kernel's table of loaded modules. The search is not sensitive to letter case. If the name is not found, it is taken as a filename; the extension .DLL is appended and the file is looked up in the disk directory specified in the *libpath=* statement in the configuration file. If it is found, the executable file is loaded from disk (and if it has the INITGLOBAL attribute its initialization routine is called).

When the module is known to be loaded, OS/2 checks to see if it was bound to the current process when that process was loaded. If not, the module's use-count is incremented and its segments are added to the LDT of the current process (and if the module has the INITINSTANCE attribute its initialization routine is called). Finally a handle is returned that can represent the loaded module in the *DosFreeModule*, *DosGetModName*, and *DosGetProcAddr* calls.

The "object name buffer" described by the first two parameters is only used when the load fails. Then it receives an ASCIIZ string that relates in some way to the failure to load. Usually it will contain a copy of the third parameter.

Discussion: *See* 4.5 **Include:** PROCESS

Related: *DosFreeModule*, *DosGetModName*, *DosGetProcAddr*, *DosGetResource*

Family API: not available
Errors to consider:
2 The file *modname*.DLL must exist in the *libpath*= directory
4 Not enough file handles, use *DosSetMaxFH* to increase them
5 *modname* DLL exists but can't be opened for input
123 The third parameter must be a filename of 1-8 characters

```
extern unsigned far pascal DOSLOADMODULE(
                        char far *,     /* object name buffer */
                        int,            /* buffer length */
                        char far *,     /* module name */
                        HANDLE far *);  /* received handle */
#define DosLoadModule DOSLOADMODULE

function DosLoadModule(vars FailString : char; { e.g. ObjName[1] }
                           FailSize : integer;
                     consts ModName : char; { e.g. ModName[1] }
                       vars TheHandle : Handle
                       ) : word; external;
```

DosLockSeg

Reaccess disposable segment

selector returned at allocation

If the reference segment has been unlocked and then discarded, an error code is returned. (In that case the segment is gone; use *DosReallocSeg* with the original size value to regain the storage space and regenerate its contents from other data.) If the segment has not been discarded, its lock reference count is incremented. Before a disposable segment can be discarded, as many *DosUnlockSeg* calls must be made as *DosLockSeg* calls. However, the counter has only 8 bits of precision, so if enough *DosLockSeg* calls are made to raise the count to 255, the segment is permanently locked and neither call will change the count again.

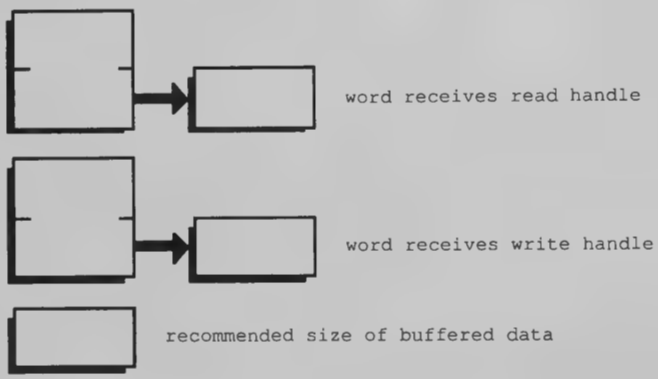
Discussion: *See* 7.3.7 **Include:** STORAGE
Related: *DosUnlockSeg, DosAllocSeg*
Family API: not available
Errors to consider:
5 The specified segment doesn't exist or wasn't allocated with the right option
157 Sorry, that segment was discarded

```
extern unsigned far pascal DOSLOCKSEG( SELECTOR );
```

```
#define DosLockSeg DOSLOCKSEG

function DosLockSeg  (TheSel : Selector
                      ) : word; external;
```

DosMakePipe

Create pipe structure



This service creates a pipe and returns its two handles, which will be unique within the current process. There must be slots for two handles in the table of file handles in the current process; *DosSetMaxFH* can be used to increase this limit. The handles returned can be duplicated using *DosDupHandle* and their attributes (particularly inheritance) can be changed with *DosSetFHandState*. The pipe will exist until both handles (and all duplicates of them including inheritances) have been closed. A pipe incorporates an internal storage buffer where *DosWrite* stores data until *DosRead* takes it. If *DosWrite* finds the buffer full it suspends the calling process until *DosRead* makes buffer space available. The buffer size in the third parameter limits the amount of data that can be written before the writing thread is suspended. For efficiency it should be at least as large as the largest single block that will be sent through the pipe, but not more than twice that size. The size may not exceed 65,536 less the size of an OS/2 internal control block which occupies 32 bytes in the first release of the system (65,280 should be safe forever).

Discussion: *See* 6.1.5 **Include:** SESSION
Related: *DosWrite, DosRead, DosQFHandState, DosSetFHandState*
Family API: not available
Errors to consider:
4 There are not two unused handles available

```
extern unsigned far pascal DOSMAKEPIPE(
                        HANDLE far *,
                        HANDLE far *,
                        unsigned int);
#define DosMakePipe DOSMAKEPIPE

function DosMakePipe  ( vars ReadHdl : Handle;
                        vars WriteHdl: Handle;
                             BuffSize: word
                        ) : word; external;
```

DosMemAvail

Get size of largest free segment

receives 32-bit number

The size of the largest contiguous block of vacant storage is returned as a 32-bit integer (it is not likely to exceed 24 bits, 16 megabytes). The number may become invalid before it is examined due to allocations by other programs. Take the result as a minimum. If *memman=move* is given in the configuration file, it may be possible to allocate a larger segment. If *memman=swap* is given, a larger segment can almost certainly be allocated.

Discussion: *See* 7.3.1 **Include:** STORAGE

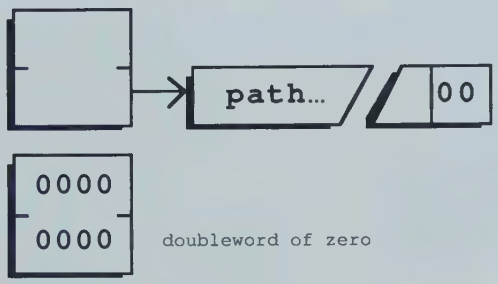
Related: *DosAllocSeg, DosAllocHuge*

Errors to consider:

```
extern unsigned far pascal DOSMEMAVAIL(long far *);
#define DosMemAvail DOSMEMAVAIL

procedure DosMemAvail  (vars MemAvail:integer4
                        ); external;
```

DosMkDir

Make a directory



A new directory is constructed. The path string may include a drive letter. If the path string doesn't specify the root, the current directory path is prefixed to it. The resulting string may be longer than 64 bytes; that is, this call can create a directory whose path *DosChDir* will not accept.

All names in the path, except the last, must exist and be directories. The last name must not presently exist. It is created as a directory. The new directory contains only two entries, the mandatory "dot" and "doubledot" entries. The "doubledot" entry will point to the parent directory, the one whose name preceded it in the path string (the root, if no name preceded it.)

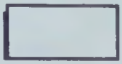
Discussion: *See* 8.6.1 **Include:** FILES
Related: *DosRmDir*
Family API: available
Errors to consider:

3	Path not found: invalid drive or a member doesn't exist or isn't a directory
5	The last name in the path exists already
87	The last parameter must be a doubleword of 0

```
extern unsigned far pascal DOSMKDIR(char far *, long);
#define DosMkDir DOSMKDIR

function DosMkDir       (vars ThePath : char; {e.g. Pathstr[1]}
                         Zero : integer4
                         ) : word; external;
```

DosMonClose

Stop monitoring a device



handle returned by DosMonOpen

If the handle was prepared by *DosMonOpen* and has not been closed, it is closed. Buffers that have been registered using *DosMonReg* with this handle are deregistered; they will no longer be part of the data stream for the device driver associated with the handle. If a thread in the process is waiting in a *DosMonRead* call on a deregistered buffer, it continues to wait and is effectively dead.

Discussion: *See* 16.3.3 **Include:** DEVICE

Related: *DosMonOpen, DosMonRegister*

Family API: not available

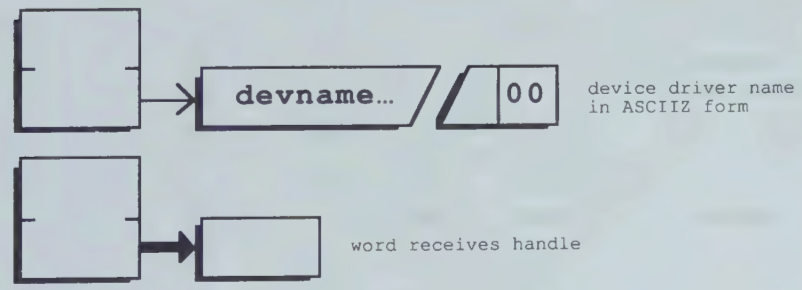
Errors to consider:

381 The handle must have been returned by *DosMonOpen* and still be open

```
extern unsigned far pascal DOSMONCLOSE(HANDLE);
#define DosMonClose DOSMONCLOSE

function DosMonClose(TheHdl : Handle):word; external;
```

DosMonOpen

Get handle for device driver to monitor



If the device name is one defined in a device driver's header, and if that device driver supports monitors, a handle is associated with the device driver and returned in the second parameter. Possible device names include "KBD$," "LPT1" and "MOUSE$." The serial ports, "COM1" and "COM2," do not support monitors. Before monitoring can begin the handle must be used with *DosMonReg* to

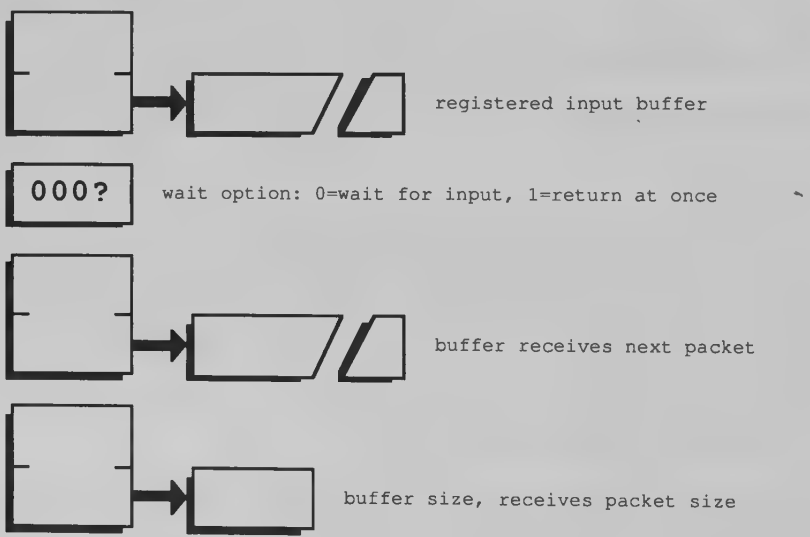
register a pair of buffers. When monitoring is complete, *DosMonClose* releases the handle.

Discussion: *See* 16.3.1 **Include:** DEVICE
Related: *DosMonClose, DosMonReg*
Family API: not available
Errors to consider:
4 No available file handles (see *DosSetMaxFH*)
110 The device named must exist and support monitors

```
extern unsigned far pascal DOSMONOPEN(
                char far *,     /* device name, asciiz */
                HANDLE far *    /* handle for use in MonRegister */
                );
#define DosMonOpen DOSMONOPEN

function DosMonOpen ( consts DevName : char; { 1st of ASCIIZ string
}
                      vars   TheHdl  : Handle
                ): word; external;
```

DosMonRead

Accept next monitor packet



The first parameter must address a buffer registered as an input buffer with DosMonReg. If no packet is ready, the second parameter is consulted: if it is 1,

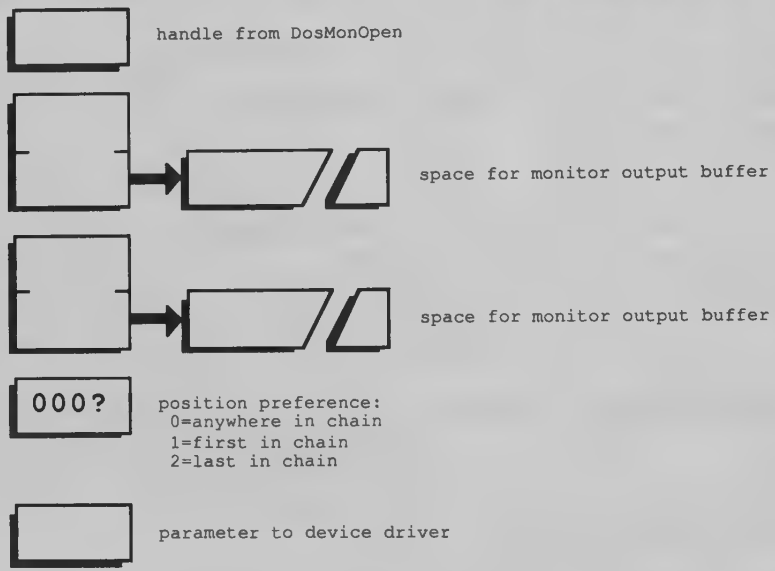
the function ends with an error code; if zero, the calling thread is suspended until a packet arrives. When a packet is ready in the buffer, its size is checked against the size specified in the fourth parameter. If there is room, the packet is copied to the receiving buffer and its actual size is set in the fourth parameter. The size and contents of a monitor data packet depends on the device driver that supplies it. Keyboard, printer, and mouse data packets are discussed in the text.

Discussion: *See* 16.3.4, 16.3.2 **Include:** DEVICE
Related: *DosMonOpen, DosMonReg, DosMonWrite*
Family API: not available
Errors to consider:
379 The wait option is not 0 or 1, or the monitor buffer is not a registered buffer, or monitoring has been ended with *DosMonClose*
382 The size given is less than the packet size
383 No packet is available and the wait option was 1

```
extern unsigned far pascal DOSMONREAD(
                struct MonBuffer far *, /* registered input buffer */
#define MONWAIT 0
#define MONNOWAIT 1
                int,            /* 0 to wait, 1 nowait */
                void far *,     /* ->returned KeyStroke/PrintData */
                int far *       /* inout bytecount */
                );
#define DosMonRead DOSMONREAD

function DosMonRead( vars BufInp : MonBuffer; { see DosMonReg }
                          TheOpt : integer;   { Mon[No]Wait }
                          TheData : ADSMEM;   { ADS
KeyStroke/PrintData }
                     vars TheSize : integer   { in/out bytecount }
                        ):word; external;
```

DosMonReg

Insert buffers into device data stream



If the first parameter is a handle prepared by *DosMonOpen* and not since closed, the buffers are examined. The first word of each buffer must contain its total size (the maximum packet size will be at least 18 bytes less); 128 bytes is sufficient for keyboard monitors while 256 is recommended for printer monitors. The two buffers must be allocated in the same data segment and must not overlap. The fourth parameter specifies where in a chain of monitors this process wants to be. When two processes ask to be first or last, the first one to register takes priority; the second takes the spot next nearer the middle of the chain. Other values (3 through 7) are accepted in this parameter but their effects are not documented. The fifth parameter is interpreted by the device driver. Its uses with the keyboard, mouse, and printer device drivers is discussed in the text. If all parameters are acceptable, the buffers are installed in the data stream for the device. Until monitoring is ended by *DosMonClose*, all data for the stream arrives at the input buffer and must be taken with *DosMonRead*; the only data to continue in the stream is that put in the output buffer with *DosMonWrite*. If the process fails to read or write the flow of data will stop.

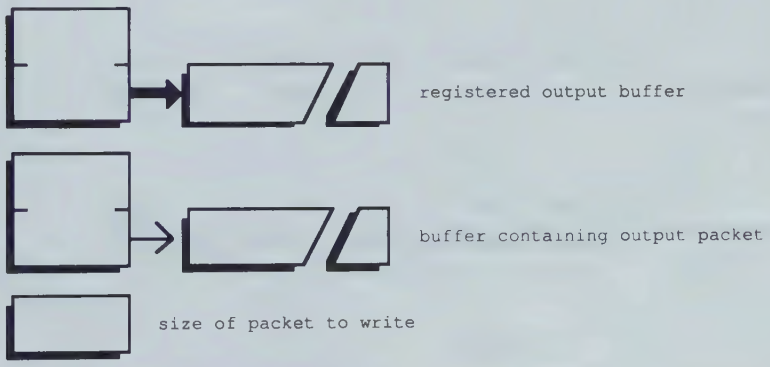
Discussion: *See* 16.3, 16.3.2 **Include:** DEVICE
Related: *DosMonOpen, DosMonClose, DosMonRead, DosMonWrite*

Family API: not available
Errors to consider:
379 Something is wrong with one of the parameters
381 The handle wasn't opened with *DosMonOpen*

```
extern unsigned far pascal DOSMONREG(
                HANDLE,          /* handle from DosMonOpen */
                struct MonBuffer far *, /* input buffer */
                struct MonBuffer far *, /* output buffer */
                int,             /* 1=first, 2=last, 0=no opinion */
                int              /* device dependent word */
                );
#define DosMonReg DOSMONREG

function DosMonReg(       TheHdl : Handle; { from DosMonOpen }
                    consts BufInp : MonBuffer; { MonRead buffer }
                    consts BufOut : MonBuffer; { MonWrite buffer }
                           ThePosn : integer; { see MonFirst, etc consts }
                           TheParm : word      { screen group usually }
                         ):word; external;
```

DosMonWrite

Pass monitor data forward



The first parameter must address a buffer registered as an output buffer by *DosMonReg*. The packet size given by the third parameter is compared to the size of the registered output buffer. If there is room, the packet of data addressed by the second parameter is copied into the output buffer (if another monitor is waiting, the data may be copied directly to its input buffer) for the size given in the third parameter. The form and content of a monitor data packet depends on the device driver that creates and receives them. Keyboard, mouse, and printer packets are discussed in the text.

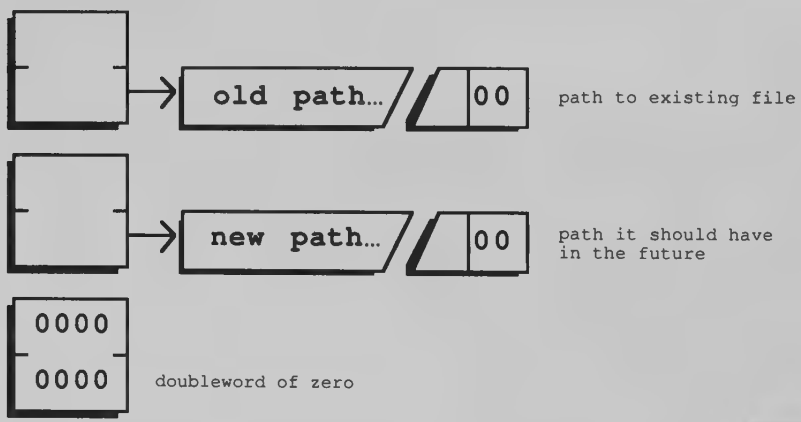
Discussion: *See* 16.3.4 **Include:** ASYNC

Related: *DosMonRead, DosMonReg*
Family API: not available
Errors to consider:
379 The first parameter isn't a registered output buffer or monitoring has been ended by *DosMonClose*
384 The packet size must be less than or equal to the size of the registered buffer

```
extern unsigned far pascal DOSMONWRITE(
               struct MonBuffer far *, /* registered output buffer */
               void far *,     /* ->output KeyStroke/PrintData */
               int             /* count of bytes */
               );
#define DosMonWrite DOSMONWRITE

function DosMonWrite(   vars BufOut : MonBuffer; { see DosMonReg }
                           TheData : ADSMEM;  { ADS KeyStroke/PrintData }
                             TheSize : integer  { sizeof above }
                        ): word; external;
```

DosMove

Rename or move file



The first path must name an existing file that is not open by another process. The second path must name existing directories but a file that does not exist. If the two paths are the same except for the filenames, the existing file is renamed. If the paths differ, the entry for the file is deleted from the first path and added to the second, under the specified name.

Discussion: *See* 8.6.2 **Include:** FILES

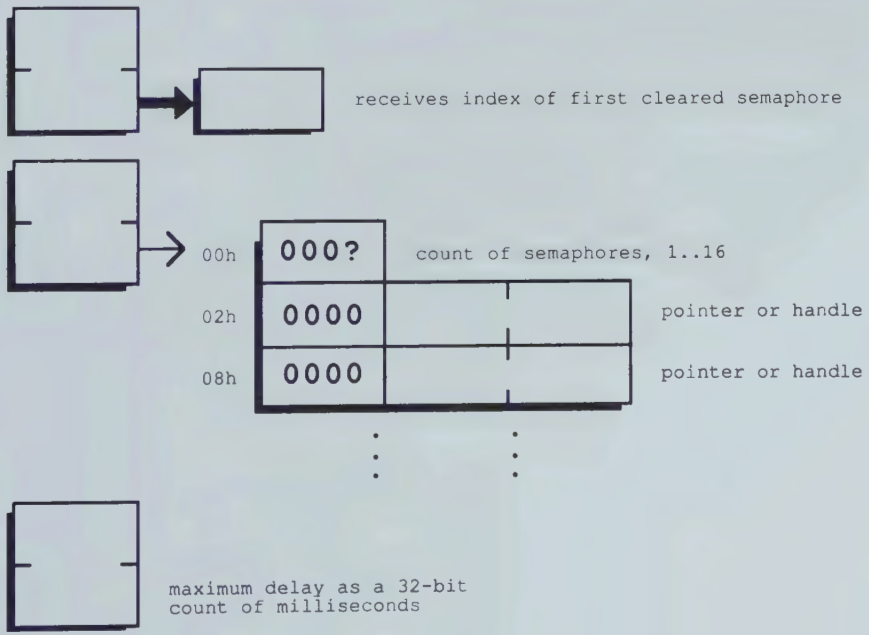
Related: *DosDelete*
Family API: available
Errors to consider:

2 File not found (but old path is OK)
3 One of the paths was no good (a member wasn't found or isn't a directory)
5 The old file is a directory or the new file already exists
17 The old and new paths must be on the same disk
32 The old file is open in another process and can't be moved

```
extern unsigned far pascal DOSMOVE(char far *, char far *, long);
#define DosMove DOSMOVE

function DosMove        (vars OldPath : char;
                         vars NewPath : char;
                              AnyZero : integer4
                         ) : word; external;
```

DosMuxSemWait

Wait until one of many semaphores cleared



This call permits a thread to wait for the first of as many as 16 events to occur. The second parameter points to a word containing a count of the semaphores followed by that many 3-word fields. Each field comprises a word that must be zero and a semaphore pointer (either a far pointer to a local semaphore or the 32-bit handle of a system semaphore). The index in this list of the first semaphore to be cleared will be returned in the word addressed by the first parameter. If any of the semaphores is initially clear, its index is noted and control returns at once. Otherwise this thread is suspended until one of the semaphores is cleared. Then control will return; the semaphore is not retested as with *DosSemWait*. The delay parameter sets an upper limit on the delay; use minus-1 (32 binary 1-bits) to allow an indefinite wait, or use zero to assure immediate return if no semaphores are clear.

Discussion: *See* 9.4.4, 9.4.5 **Include:** ASYNC
Related: *DosSemSet, DosSemClear, DosSemWait*
Family API: not available
Errors to consider:

6 All handles must be returned by *DosOpenSem* or *DosCreateSem* or be valid far pointers
121 The specified time elapsed and no semaphore cleared
151 The event count must be between 1 and 16
152 The system limit on concurrent mux-waits has been exceeded
153 The list format is invalid (reserved word not zero?)

```
extern unsigned far pascal DOSMUXSEMWAIT(
                        int far *,
                        struct MUXWAITLIST far *,
                        TIMEINTERVAL);
#define DosMuxSemWait DOSMUXSEMWAIT

function DosMuxSemWait (vars WhichOne: integer;
                        vars TheList: MuxWaitList;
                             MaxTime: TimeInterval
                        ): word; external;
```

DosNewSize

Change file's size

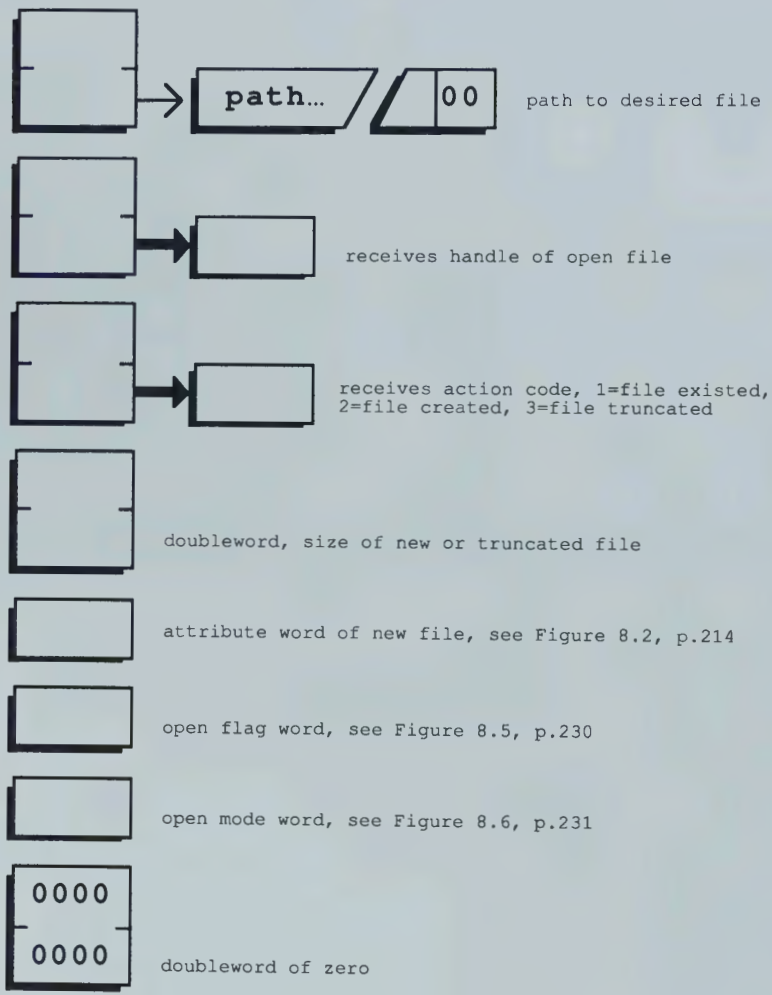
The file is extended or truncated to have the size specified. The file's end of data size will be set exactly; its allocated size will be set to the next higher allocation unit. If the file is made longer, the contents of the new space are unpredictable.

Discussion: *See* 8.9, 8.7.4 **Include:** FILES
Related: *DosOpen, DosChgFilePtrr*
Family API: available
Errors to consider:
6 The handle must be open to a disk file
112 Insufficient disk space to resize file

```
extern unsigned far pascal DOSNEWSIZE(HANDLE, FILEOFFSET);
#define DosNewSize DOSNEWSIZE

function DosNewSize     (TheHandle : Handle;
                         TheNewSize: integer4
                         ) : word; external;
```

DosOpen

Open a file



One of several actions is taken depending on the Open Flag parameter and on the file specified by the path. If the action succeeds, a handle is returned to designate the opened file in other calls. The sharing mode in the Mode word determines whether other processes will be able to use the same file concurrently. The opened file is identified with this process and will be closed if the process ends

(an explicit *DosClose* is strongly recommended).

If the file exists and the Flag calls for opening existing files, the file is opened for input or output, as called for in the Mode word. The action word returned is 1. In this case the file named may in fact be a device such as "PRN," "COM2," "KBD$," or "SCREEN$." If the file exists and the Flag calls for truncating an existing file, the file's size is changed (as in *DosNewSize*) and the file is opened. If the file does not exist and the Flag calls for creating a new file, the file is created with the attribute and size given, and opened. In these cases the file specified must be a disk file. The returned action word is either 2, created, or 3, truncated. OS/2 guarantees mutual exclusion: when concurrent processes try to create the same file, only one process will succeed in making the file and all others will find it to be an existing file. When concurrent processes attempt to open a file with conflicting sharemodes, one will succeed and all others will be denied.

When the Mode specifies "DASDOpen" and the Flag specifies "open existing file" and the path is only a drive letter and colon, that disk volume is opened as a file with the access and sharing options given. The file's contents begin with the first sector of the first track and run in sector and track order to the end of the physical volume.

Discussion: *See* 8.7 **Include:** FILES

Related: *DosClose, DosDupHandle, DosQHandType, DosQFHandState, DosSetFHandState, DosQFileMode, DosSetFileMode, DosRead, DosReadAsync, DosWrite, DosWriteAsync, DosChgFilePtr, DosBufReset, DosNewSize, DosFileLock*

Family API: Access and sharing modes only checked if "share" support is active. DASDOpen does not give exclusive access to the device.

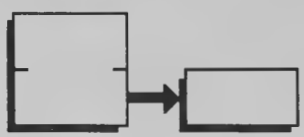
Errors to consider:

2	File not found (but path is OK)
3	Path not found: invalid drive or a member doesn't exist or isn't a directory
4	No available file handles
5	The specified file is a directory, or read-only, or the attribute parameter contains unsupported nonzero bits
12	The access or share values in Mode word are improper
32	Another process has the file open with a conflicting access mode
87	The Flag or Mode word is improper or the final doubleword is nonzero
110	Failure as requested by Flag word (file exists or doesn't)
112	Not enough disk space to create/resize file
123	Filename or filetype is malformed
203	Filename or filetype is malformed
206	Filename or filetype is too long

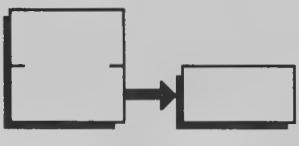
```
extern unsigned far pascal DOSOPEN(
                char far *,      /* path */
                HANDLE far *,    /* handle */
                int far *,       /* action */
                long,            /* new/trunc file size */
                ATTRWORD,        /* new file attr */
                OPENFLAG,        /* sum of OF_values */
                OPENMODE,        /* sum of OM_values */
                long);           /* must be 0l */
#define DosOpen DOSOPEN

function DosOpen        (consts ThePath : char;      { path }
                         vars   TheHandle : Handle; { handle }
                         vars   TheAction : integer;{ action taken }
                                TheSize : integer4; { size if new/trunc }
                                TheAttrs : AttrWord;{ attr of new file }
                                TheFlag : OpenFlag; { sum of OF values }
                                TheMode : OpenMode; { sum of OM values }
                                AnyZero : integer4 { must be zero }
                         ) : word; external;
```

DosOpenQueue

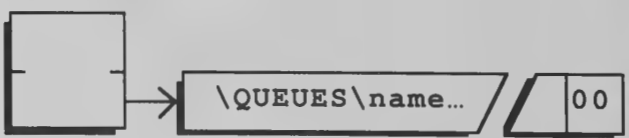
Gain access to queue



word receives Process ID of queue's owner



word receives handle of queue



ASCIIZ name of queue

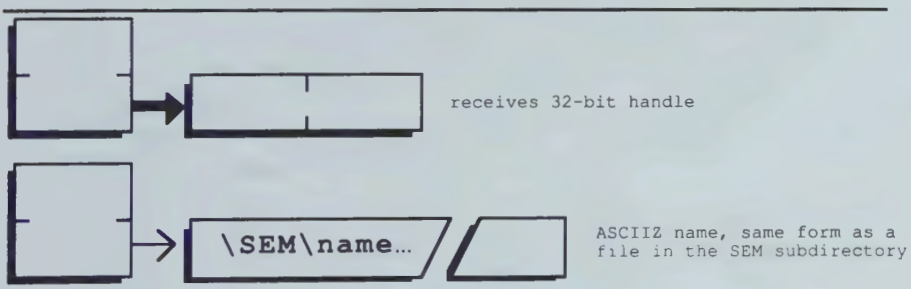
The third parameter specifies a name in the form of a filename in an imaginary directory "QUEUES"; that is, the name must begin with "\QUEUES\" and continue with a properly formatted filename and optional file extension. If no queue of the specified name exists an error code is returned. The ID of the process that created the queue is returned in the first parameter and a handle for the queue in the second. The queue handle may be used in *DosWriteQueue* calls. The process ID may be used in *DosFlagProcess* calls to signal the queue's owner.

Discussion: *See* 9.7, 9.7.2 **Include:** QUEUES
Related: *DosCloseQueue, DosCreateQueue, DosWriteQueue*
Family API: not available
Errors to consider:
335 The queue name must begin "\QUEUES\" and end in a valid filename
343 No queue of that name exists

```
extern unsigned far pascal DOSOPENQUEUE(
                        PROCESSID far *,
                        HANDLE far *,
                        char far *);
#define DosOpenQueue DOSOPENQUEUE

function DosOpenQueue(  vars OwnerPID : ProcessID;
                        vars TheHdle  : Handle;
                      consts TheName  : char { e.g. QueNameStr[1] }
                        ):word; external;
```

DosOpenSem

Access a system semaphore



If a semaphore of this name exists, a handle for it is returned (note that the handle is 32 bits unlike the common 16-bit file handle). The handle may be used in any semaphore operation. The handle may be passed to a child process as well.

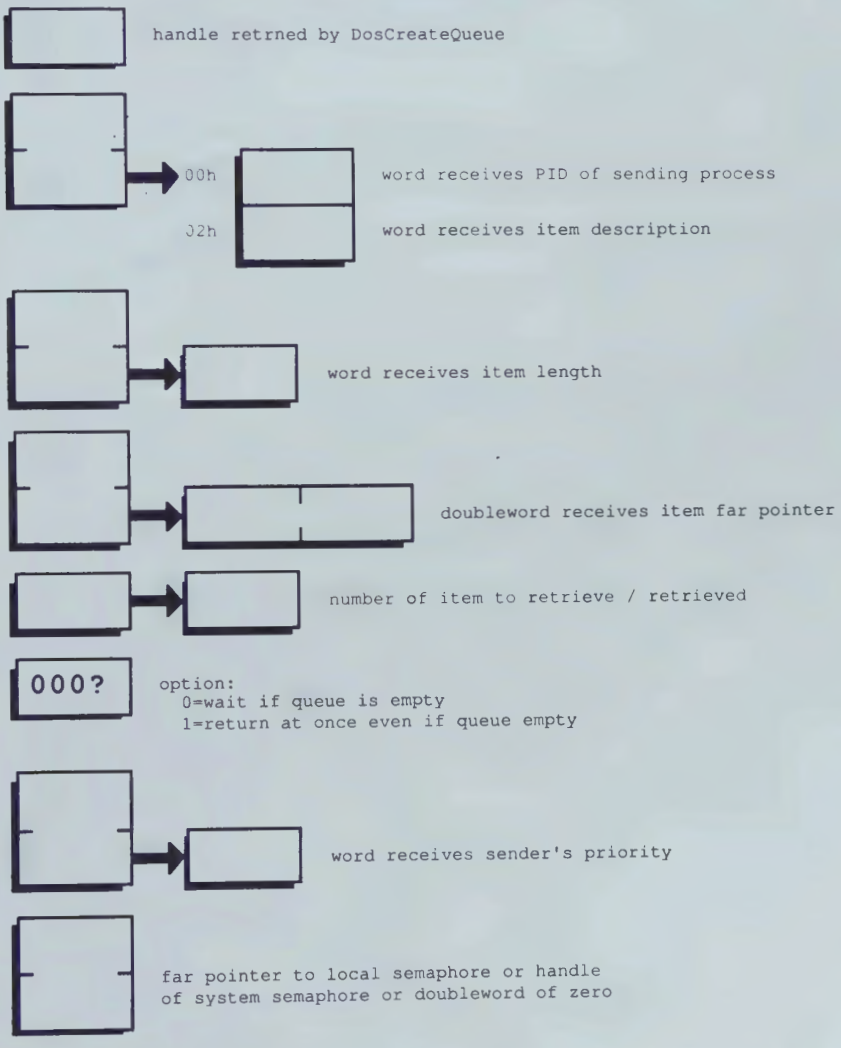
Discussion: *See* 9.4.5 **Include:** ASYNC

Related: *DosCreateSem, DosCloseSem*
Family API: not available
Errors to consider:
100 The system limit on open semaphores has been exceeded
106 The system limit on processes per semaphore has been exceeded
123 The name must be an ASCIIZ string of the form "\SEM\filename.ext" using only characters valid in a filename
187 No semaphore of that name can be found (create it instead)

```
extern unsigned far pascal DOSOPENSEM(  SEMHANDLE far *,
                                        char far *);

#define DosOpenSem DOSOPENSEM

function DosOpenSem     (vars TheHdle: SemHandle;
                         vars TheName: char { e.g. AString[1] }
                         ) : word; external;
```

DosPeekQueue

Get element from queue



Unless the first parameter specifies a queue created by the current process and not yet closed, an error code is returned. Otherwise the queue is tested. If it is empty, the option in the sixth parameter is sampled: if is zero, the calling thread is suspended until *DosWriteQueue* is used to put an item in the queue; if it is 1,

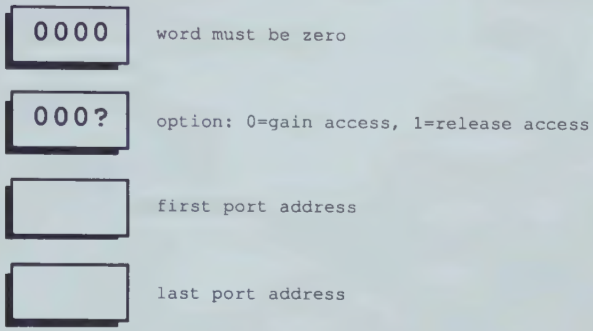
an error code is returned and, if the last parameter is a semaphore address or handle, that semaphore will be cleared when the next item is put in the queue. When the queue is not empty, the number in the fifth parameter is examined. If it is zero, the head item is selected. Otherwise the number must have been returned by a prior call to *DosPeekQueue*, and the item that follows that position is selected. The position of the selected item is set in the fifth parameter so that, in a FIFO queue, repeated calls to *DosPeekQueue* will return all items in order. In a LIFO or priority-ordered queue new items can be inserted ahead of the position being Peeked and will not be selected until the scan starts over. The fields of the selected item are distributed into the second, third, fourth and seventh parameters, as for *DosReadQueue*. The item remains in the queue; use *DosReadQueue* to remove it.

Discussion: *See* 9.7.3, 9.7.5 **Include:** QUEUES
Related: *DosCreateQueue, DosReadQueue, DosWriteQueue*
Family API: not available
Errors to consider:
330 Only the queue's owner is permitted to do this
333 There is no queue item following the number given in the fifth parameter (end of queue)
337 The first parameter must be the handle of an existing queue
342 The queue is empty (and the wait option is 1)

```
extern unsigned far pascal DOSPEEKQUEUE(
                        HANDLE,
                        struct PIDDESC far *, /* PID, desc word */
                        unsigned far *,  /* length */
                        void far * far *,/* address */
                        int far *,       /* item position or zero */
                        int,             /* wait option */
                        int far *,       /* priority */
                        SEMHANDLE);
#define DosPeekQueue DOSPEEKQUEUE

function DosPeekQueue(     TheHdle : Handle;
                       vars DescNPid: PIDDesc;
                       vars TheSize : word;
                       vars ThePtr  : ADSMEM; { right, vars...ADSMEM }
                       vars ThePosn : integer;
                            WaitOpt : integer;
                       vars ThePrio : integer;
                            TheSem  : SemHandle { use NilADS for nil }
                         ):word; external;
```

DosPortAccess

Request access to I/O addresses



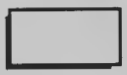
The call should be issued from a code segment with I/O privilege (linked with the IOPL keyword). When the second parameter is zero, the program is requesting the right to issue IN and OUT instructions to the I/O addresses in the range specified. When the second parameter is 1, the program is giving up access to those ports (which it must previously have been given access to). In present levels of OS/2 running in an 80286, I/O privilege is a blanket right that is conferred automatically on code linked with the IOPL keyword; the hardware lacks the ability to discriminate between port numbers in I/O instructions. Thus in present systems this call is only a formality. However, in future levels of the system on more elaborate hardware, the system may have the ability to enforce the requirement. Since the call takes almost no time to execute, use it now in order to be portable later.

Discussion: none **Include:** DEVICE
Related: *Dos*
Family API: available
Errors to consider: none

```
extern void     far pascal DOSPORTACCESS(int, /* must be zero */
                                         int, /* 0 or 1 */
                                         unsigned, /* first */
                                         unsigned); /* last */
#define DosPortAccess DOSPORTACCESS

procedure DosPortAccess( AnyZero : integer; { must be 0 }
                         Option : integer;  { 0=grant, 1=release }
                         FirstPort : word;
                         LastPort : word
                        ):word; external;
```

DosPurgeQueue

Empty a queue



Unless the first parameter specifies a queue created by the current process and not yet closed, an error code is returned. Otherwise the queue is made empty, any elements in it being discarded.

Discussion: *See* 9.7.7 **Include:** QUEUES

Related: *DosCreateQueue, DosQueryQueue*

Family API: not available

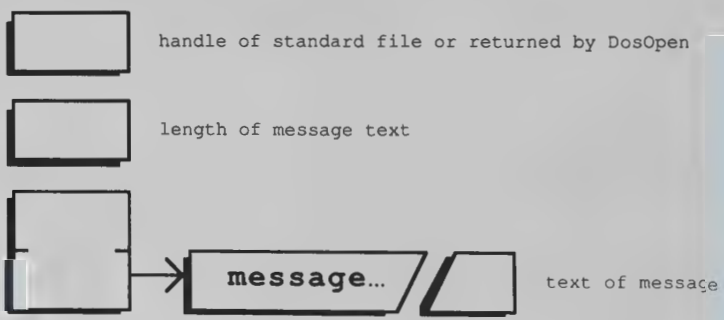
Errors to consider:

337 The handle must be of an existing queue created by this process

```
extern unsigned far pascal DOSPURGEQUEUE(HANDLE);
#define DosPurgeQueue DOSPURGEQUEUE

function DosPurgeQueue(TheHdle:Handle
                       ):word; external;
```

DosPutMessage

Write message string to handle



A string of bytes is written to a file, as for *DosWrite*. The file handle may be one returned by *DosOpen*, or one of the standard handles such as 1, stdout. This service assumes that the string contains ASCII text, and that the destination is a

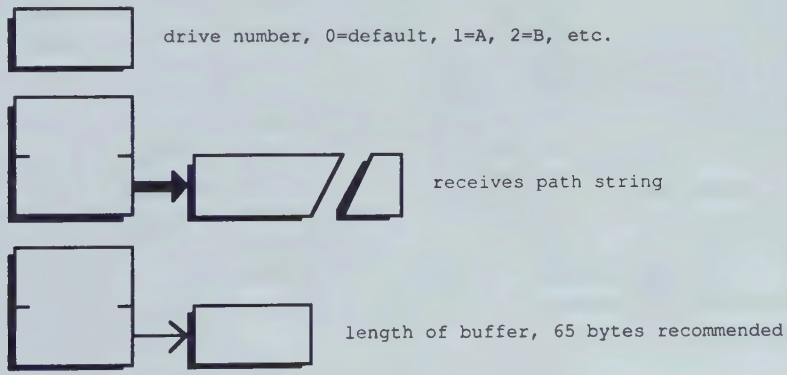
device with an 80-byte width (a screen or printer), and that output begins in column 1. If the length exceeds 80 bytes, the line is broken at a word boundary, that is, at the space character nearest to the 80-byte limit. If no space can be found in a stretch of 80 bytes, the line is broken after the 78th byte.

Discussion: *See* 5.4.6 **Include:** PROCESS
Related: *DosGetMessage, DosInsMessage, DosWrite, DosOpen*
Family API: available
Errors to consider: none

```
extern void far pascal DOSPUTMESSAGE(
                HANDLE,
                int,
                char far *);
#define DosPutMessage DOSPUTMESSAGE

procedure DosPutMessage(        TheHandle : Handle;
                                TheSize : word;
                        consts TheMsg : char
                        ); external;
```

DosQCurDir

Get current directory pathstring



The path string for the current directory for the specified drive is returned as an ASCIIZ string. The current directory for each drive is stored on a per-process basis.

If the specified drive has demountable volumes, the system may try to read the label of the current volume; if there's no disk in the drive not-ready error will be reported to the operator.

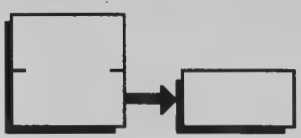
The path string is at most 64 bytes (plus a null byte terminator). It does not contain a drive-letter. It does not begin with a backslash. It does not end with a backslash. If the current directory for the drive is the root, the returned string is null.

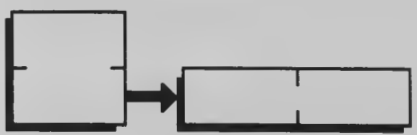
Discussion: *See* 8.5.3 **Include**: FILES
Related: *DosChDir*
Family API: available
Errors to consider:
15 The drive number must exist in the system
111 The buffer is too short for the result

```
extern unsigned far pascal DOSQCURDIR(
                        int, /* drive number */
                        char far *, /* output area */
                        int far * /* buffer size */
                        );
#define DosQCurDir DOSQCURDIR

function DosQCurDir     (TheDrive : integer;
                         vars ThePath : char; { e.g. PathStr[1] }
                         consts PathSize : integer
                         ) : word; external;
```

DosQCurDisk

Get default disk letter and drive map





The number of the default drive, which is assumed in all pathnames that don't start with an explicit drive letter, is returned. Also returned in a 32-bit doubleword is a 25-bit "map" of the drives that exist in the system. Each bit from bit 0 to bit 25 corresponds to a drive from letter A to letter Z, and a 1-bit

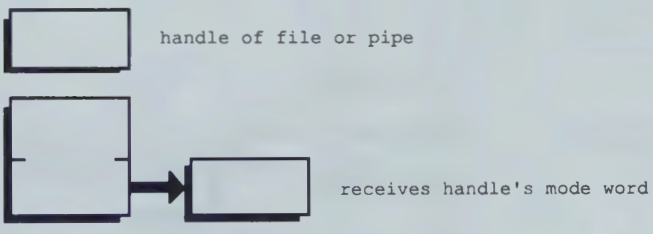
means the drive exists and would be valid for the *DosSelectDisk* call.

Discussion: *See* 8.5.2 **Include**: FILES
Related: *DosSelectDisk, DosQFSInfo*
Family API: available
Errors to consider: none

```
extern void far pascal DOSQCURDISK(
                        int far *, /* drive number */
                        long far * );/* drive map */

#define DosQCurDisk DOSQCURDISK

procedure DosQCurDisk  (vars Drive : integer;
                        vars DriveMap : integer4
                        ); external;
```

DosQFHandState

Get file handle mode word



The Mode word of the handle is returned for examination. The Mode word of a handle establishes its access mode, share options, and other options (see Figure 8.6, p. 231). It is originally set as a parameter of *DosOpen* and may be altered by *DosSetFHandState*.

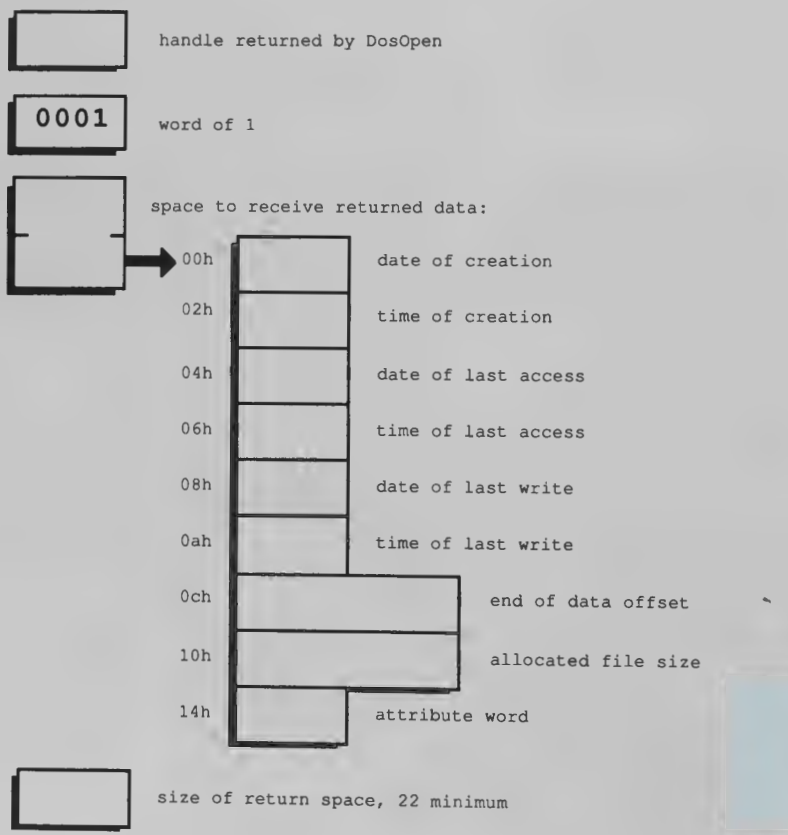
Discussion: *See* 8.10.2, 8.7.3 **Include:** FILES
Related: *DosOpen, DosSetFHandState, DosQHandType*
Family API: not available
Errors to consider:
6 The handle must be one returned by *DosOpen* and not yet closed

```
extern unsigned far pascal DOSQFHANDSTATE(
                HANDLE,
                unsigned far *);
#define DosQFHandState DOSQFHANDSTATE
```

```
function DosQFHandState(     TheHandle : Handle;
                         vars TheMode : OpenMode
                         ) : word; external;
```

DosQFileInfo

Get timestamps and sizes of open file



The timestamps, file sizes, and attribute word recorded in the directory entry of the open file are returned for examination. Some fields are not maintained in the current level of the system. The date and time of last write are maintained; use them as substitutes if other dates and times are zero. The allocated file size may be zero; round the end of data size up to a multiple of 512 bytes for an

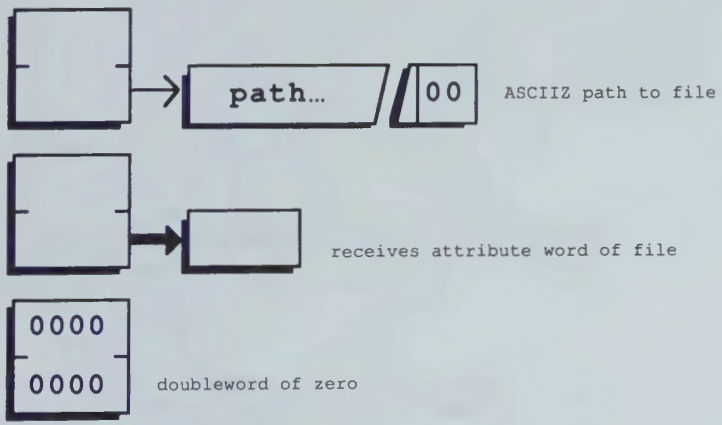
approximate size.

Discussion: *See* 8.6.6 and Figure 8.4, p. 224 **Include:** FILES
Related: *DosFindFirst, DosSetFileInfo, DosQFileMode, DosSetFileMode*
Family API: available
Errors to consider:
6 The handle must be open to a disk file
111 The buffer size parameter must be at least 22
124 The second parameter must be 1

```
extern unsigned far pascal DOSQFILEINFO(
                        HANDLE,
                        int, /* must be 1 */
                        struct FileInfo far *,
                        int /* sizeof(struct FileInfo) */
                        );
#define DosQFileInfo DOSQFILEINFO

function DosQFileInfo  (     TheHandle : Handle;
                             TheLevel  : integer; {must be 1}
                        vars TheData   : FileInfo;
                             TheSize   : integer { sizeof(FileInfo)
}
                        ) : word; external;
```

DosQFileMode

Get attribute word of file



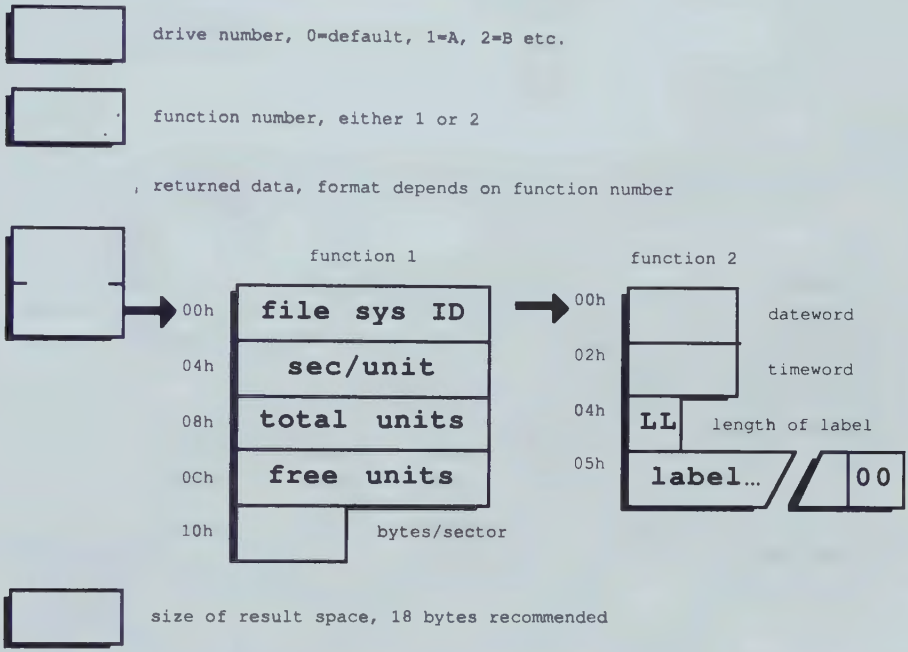
The attribute word from the directory entry of the specified file is returned. See Figure 8.2 p. 214 for the format of the attribute word.

Discussion: *See* 8.6.4, 8.3, Figure 8.2, p. 214 **Include:** FILES
Related: *DosSetFileMode, DosQFileInfo*
Family API: available
Errors to consider:
2 File not found (but path is OK)
3 Path not found: invalid drive or a member doesn't exist or isn't a directory

```
extern unsigned far pascal DOSQFILEMODE(
                        char far *,     /* path */
                        ATTRWORD far *, /* attr */
                        long);          /* zero */
#define DosQFileMode DOSQFILEMODE

function DosQFileMode  (vars ThePath : char;
                        vars TheAttr : AttrWord;
                             AnyZero : integer4
                        ) : word; external;
```

DosQFSInfo

Get information about a disk volume



One of two data structures is returned, depending on the function requested. The first, requested by function #1, contains a "file system ID number" which is zero for the basic system compatible with MS-DOS and PC-DOS, nonzero for an optional file system. It also contains numbers describing the space allocation on the volume: number of bytes in a sector (which may or may not correspond to a physical disk sector); number of sectors in an allocation unit, total units and free units. The data structure returned for function #2 contains the volume label string and the date and time it was written.

Discussion: *See* 8.5.4, 8.5.5 **Include:** FILES

Related: *DosSetFSInfo, DosQCurDisk, DosSelectDisk*

Family API: available

Errors to consider:

15 The drive number must exist in the system

111 The buffer size must be at least 18 for function 1, at least 16 for function 2

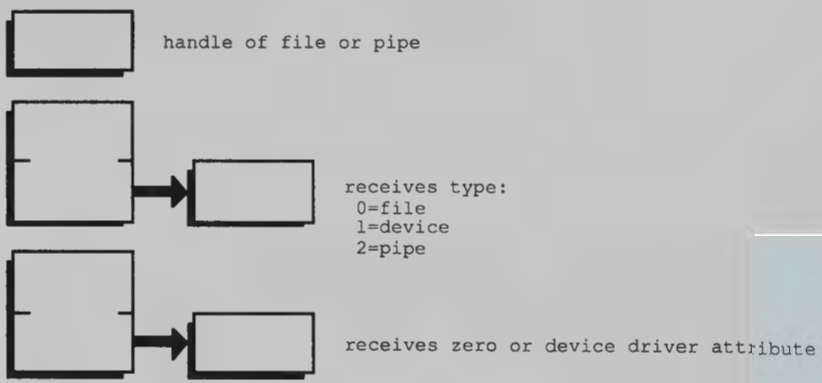
124 The function number must be 1 or 2

125 That volume has no label (function 2)

```
extern unsigned far pascal DOSQFSINFO(
                        int, /* drive number */
                        int, /* function number */
                        union FSInfo far *,
                        int /* s.b. sizeof(union FSInfo) */
                        );
#define DosQFSInfo DOSQFSINFO

function DosQFSInfo     (       Drive : integer;
                                Funcno: integer;
                        vars Buffer : FSInfo;
                        const Buflen: integer { sizeof(FSInfo) ok }
                        ) : word; external;
```

DosQHandType

Get type of file from handle



If the handle is associated with a disk file, both words are set to zero. If it is associated with a device, the type word is set to 1 and the second word is set to the device-driver attribute word (values discussed in text).

Discussion: *See* 8.10.2 **Include:** FILES
Related: *DosQFHandState*
Family API: not available
Errors to consider:

6 The handle must be one returned by *DosOpen* and not yet closed

```
extern unsigned far pascal DOSQHANDTYPE(
```

```
                        HANDLE,
                        int far *,
                        unsigned far *);
#define DosQHandType DOSQHANDTYPE

function DosQHandType  (     TheHandle:Handle;
                        vars TheType:integer;
                        vars DevAttr:word
                        ) : word; external;
```

DosQVerify

Get state of write-verify switch

word receives 0000 or 0001

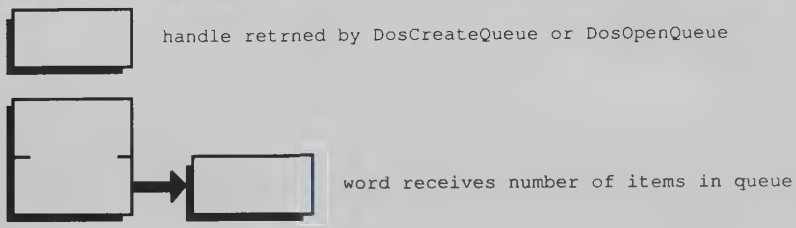
The state of the write-verify switch for this process is returned as a word of 0 (off) or 1 (on).

Discussion: *See* 8.5.1 **Include:** FILES
Related: *DosSetVerify*
Family API: available
Errors to consider: none

```
extern void far pascal DOSQVERIFY(int far *);
#define DosQVerify DOSQVERIFY

procedure DosQVerify   (vars Switch:integer
                        ); external;
```

DosQueryQueue

Inquire number of items in queue



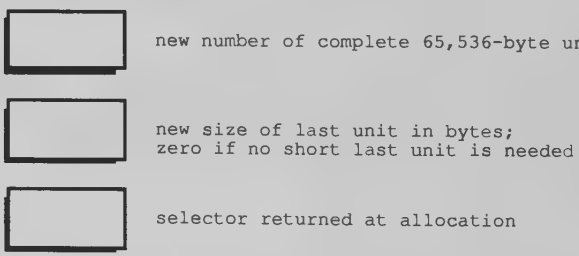
The number of items now in the specified queue is returned in the second parameter.

Discussion: *See* 9.7.7 **Include:** QUEUES
Related: *DosCreateQueue, DosOpenQueue*
Family API: not available
Errors to consider:
337 The handle must specify an open, existing queue

```
extern unsigned far pascal DOSQUERYQUEUE(HANDLE, int far *);
#define DosQueryQueue DOSQUERYQUEUE

function DosQueryQueue(      TheHdle : Handle;
                         vars Answer : integer
                         ):word; external;
```

DosReAllocHuge

Change size of huge allocation



A block of storage allocated by *DosAllocHuge* is resized to comprise more or

fewer 64K segments and/or a larger or smaller last segment. The new number of segments may not exceed the maximum promised when the block was allocated. Space is added or removed at the high-address end. The contents of the low-address portion are unchanged. There is unlikely to be a delay when the block is shrunk, but when it is made larger there may be a delay for moving or swapping in order to make room.

Discussion: *See* 7.3.3, 7.3.6 **Include:** STORAGE

Related: *DosAllocHuge, DosFreeSeg*

Family API: Size of last unit is rounded up to 16-byte multiple

Errors to consider:

5 The specified segment must be the first of a huge block

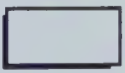
8 Not enough vacant storage

87 The new size may not be larger than the maximum set with *DosAllocHuge*

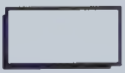
```
extern unsigned far pascal DOSREALLOCHUGE(
                        int,
                        int,
                        SELECTOR);
#define DosReallocHuge DOSREALLOCHUGE

function DosReAllocHuge(NumSegs: word;
                        EndSize: word;
                        TheSelector: Selector
                        ) : word; external;
```

DosReAllocSeg

Change size of normal allocation



new block size, zero means 65,536



selector returned at allocation

A block of storage allocated with *DosAllocSeg* is resized. Space is added or removed at the high-address end. The contents of the low-address portion are unchanged. There is unlikely to be a delay when the block is shrunk, but when it is made larger there may be a delay for moving or swapping in order to make room. Any nonhuge segment of the address space may be resized, including code and data segments created when the program was loaded. See *DosSubSet* for special restrictions on segments that have been set up for suballocation. If the referenced segment was allocated with the disposable option, reallocating it will set its locking reference count to 1, that is, locked once.

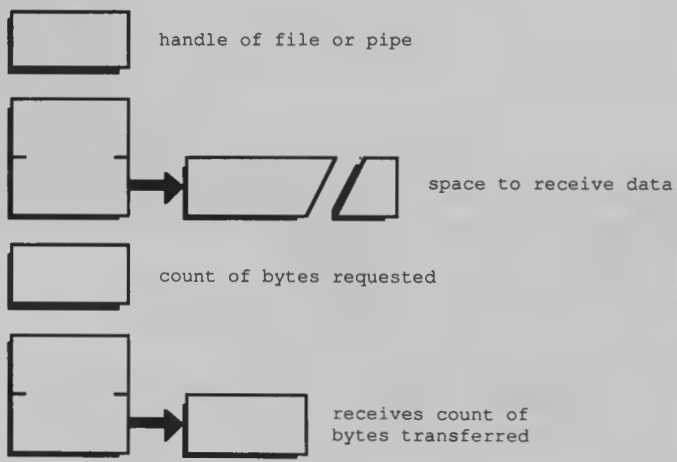
Discussion: *See* 7.3.2, 7.3.6 **Include:** STORAGE

Related: *DosAllocSeg, DosFreeSeg, DosLockSeg, DosReallocHuge*
Family API: available
Errors to consider:
5 The specified segment doesn't exist
8 Not enough vacant storage

```
extern unsigned far pascal DOSREALLOCSEG(unsigned, SELECTOR);
#define DosReallocSeg DOSREALLOCSEG

function DosReAllocSeg (NewSize: word;
                        TheSelector: Selector
                        ) : word; external;
```

DosRead

Read data from a file or pipe



Data is transferred from the file or pipe into the buffer, up to the number requested, the end of the file, or some device limit. A request for zero bytes is not an error; it simply causes zero bytes to be transferred.

Errors are signalled by a nonzero error code. The count of bytes transferred is returned in the fourth parameter; it will be zero if there was an error. The read-write pointer for the file is incremented by this count, which may be less than the count requested. A return count of zero with no error indicates (for a disk file) that the read-write pointer is at or beyond the end of file. For a device or pipe, it indicates that no more data are available at this time.

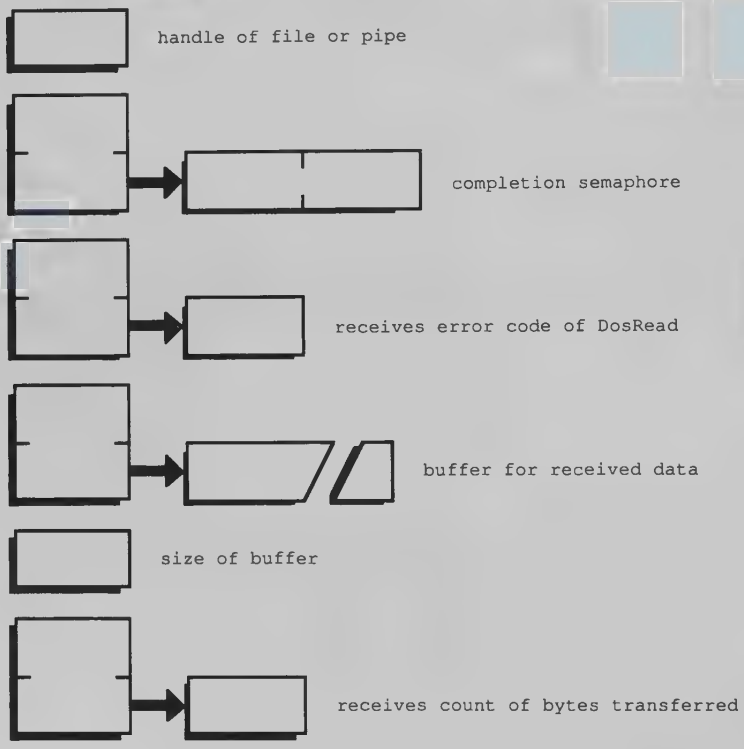
Discussion: *See* 8.11.1, 8.11.2 **Include:** FILE$

Related: *DosReadAsync, DosOpen, DosChgFilePtr*
Family API: available
Errors to consider:
6 The handle must be one returned by *DosOpen* and not yet closed
33 The file is locked in the range that would be read
(The following are seen only if the file was opened with the Return-errors option)
21 Device not ready
22 Device driver does not do reads (e.g. MOUSE$)
23 Read check
25 Error on seek to track
27 Error on search to sector
30 Error on read
31 General failure reported by device driver
34 Disk volume has been changed since file was opened

```
extern unsigned far pascal DOSREAD(
                        HANDLE,
                        char far *,
                        unsigned,
                        unsigned far *);
#define DosRead DOSREAD

function DosRead        (TheHandle : Handle;
                         TheBuffer : ADSMEM; { e.g. "ADS BufArray[1]" }
                         SizeAsked : word;
                    vars SizeMoved : word
                         ) : word; external;
```

DosReadAsync

Read data asynchronously



The system creates a thread to run asynchronously and perform a *DosRead* operation. If the thread is created successfully, a code of zero is returned and the calling thread may process other things while the read is being done. The *DosRead* operation updates the buffer area and sets the count of bytes transferred. When it is finished, its return code is deposited in the third parameter. Then the semaphore is cleared and the created thread terminates. The proper sequence of operations is:

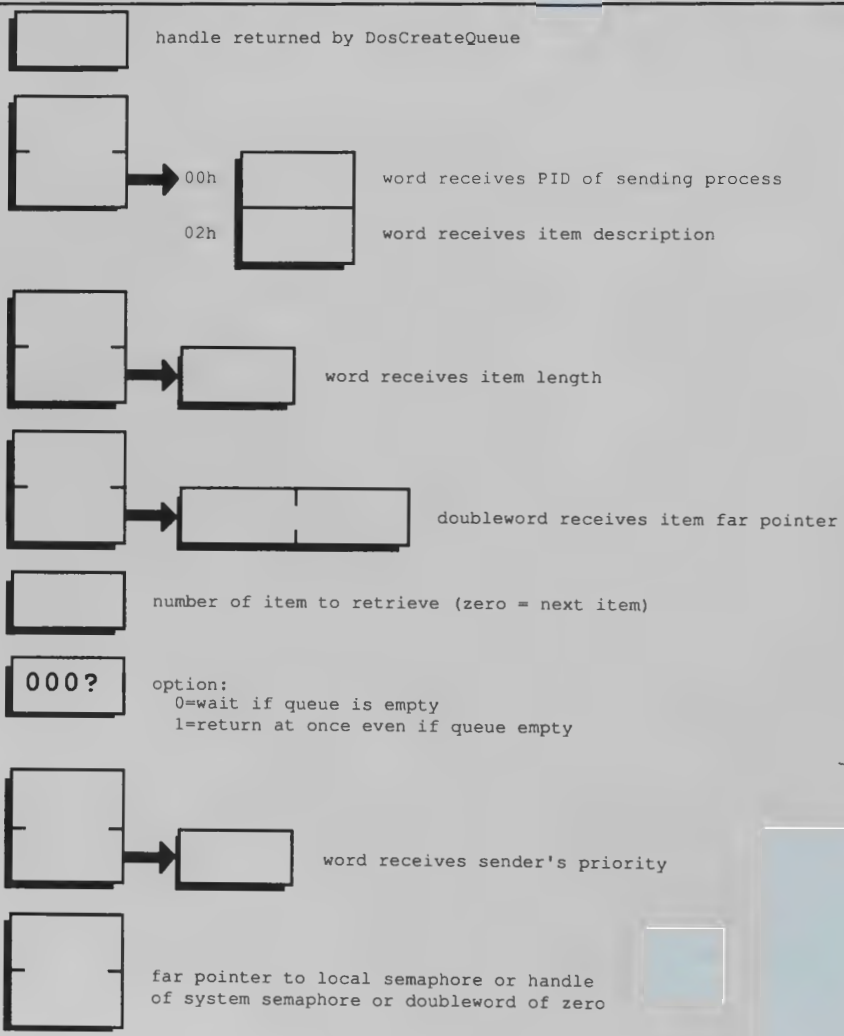
- *DosSemSet*
- *DosReadAsync*
- Do work not involving the buffer space
- *DosSemWait*
- Examine return codes and count

Discussion: *See* 8.11.5, 9.4.4 **Include:** FILES and ASYNC
Related: *DosRead, DosOpen, DosSemSet, DosSemWait*
Family API: not available
Errors to consider:
155 Not able to create a thread

```
extern unsigned far pascal DOSREADASYNC(
                        HANDLE,
                        SEMHANDLE,
                        unsigned far *, /* retcode of DosRead */
                        char far *,     /* buffer */
                        unsigned,       /* bufsize */
                        unsigned far *  /* count read */
                        );
#define DosReadAsync DOSREADASYNC

function DosReadAsync  (     TheHandle : Handle;
                             ASemaphore: SemHandle; { e.g. ADS MySem }
                        vars ReadsRet  : word; { error code of DosRead }
                             TheData   : ADSMEM; { e.g. ADS BufSpace[1] }
                             SizeAsked : word;
                        vars SizeMoved : word { size of buffer space }
                        ) : word; external;
```

DosReadQueue

Get element from queue



Unless the first parameter specifies a queue created by the current process and not yet closed, an error code is returned. Otherwise the queue is tested. If it is empty, the option in the sixth parameter is sampled: if is zero, the calling thread is suspended until *DosWriteQueue* is used to put an item in the queue; if it is 1,

an error is returned and, if the last parameter is a semaphore address or handle, that semaphore will be cleared when next an item is put in the queue. When the queue is not empty, the fifth parameter is sampled. If it is zero, the head queue item is returned according to the queue discipline, whether LIFO, FIFO or priority ordered. If the fifth parameter is nonzero it must be a number returned by *DosPeekQueue*, and that item will be returned. The returned item is removed from the queue and distributed into the second, third, fourth and seventh parameters. Most fields are the identical data specified with the *DosWriteQueue* call. The second parameter receives the item descriptor and also the Process ID number of the process that wrote the item (this may be used with *DosFlagProcess* to signal the other process). The description and size words and the far pointer are simply data; they may or may not serve those functions depending on the application.

Discussion: *See* 9.7, 9.7.6 **Include:** QUEUES

Related: *DosCreateQueue, DosPeekQueue, DosWriteQueue*

Family API: not available

Errors to consider:

330 Only the queue's owner is permitted to do this

333 There is no queue item at the number given in the fifth parameter (number not from *DosPeekQueue*, or item already read)

337 The first parameter must be the handle of an existing queue

342 The queue is empty (and the wait option is 1)

```
extern unsigned far pascal DOSREADQUEUE(
                        HANDLE,
                        struct PIDDESC far *, /* PID, desc word */
                        unsigned far *,  /* length */
                        void far * far *,/* address */
                        int,             /* item position or zero */
                        int,             /* wait option */
                        int far *,       /* priority */
                        SEMHANDLE);
#define DosReadQueue DOSREADQUEUE

function DosReadQueue(     TheHdle : Handle;
                       vars DescNPid: PIDDesc;
                       vars TheSize : word;
                       vars ThePtr  : ADSMEM; { right, vars...ADSMEM }
                            ThePosn : integer;
                            WaitOpt : integer;
                       vars ThePrio : integer;
                            TheSem  : SemHandle { use NilADS for nil }
                        ):word; external;
```

DosResumeThread

Restart suspended thread

thread ID of suspended thread

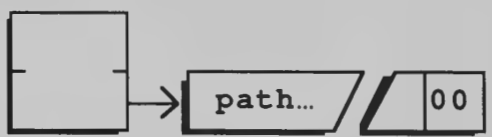
If the specified thread is part of the current process, and if it was suspended by *DosSuspendThread*, it is resumed. It will receive its normal turn to use the CPU according to its priority and class.

Discussion: *See* 9.13 **Include:** ASYNC
Related: *DosSuspendThread, DosCreateThread, DosSetPrty*
Family API: not available
Errors to consider:
309 The thread must be part of this process, and suspended

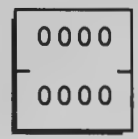
```
extern unsigned far pascal DOSRESUMETHREAD(THREADID);
#define DosResumeThread DOSRESUMETHREAD

function DosResumeThread( TheThrd : ThreadID
                        ) : word; external;
```

DosRmDir

Delete a directory



path to target directory



doubleword of zero

An existing directory is erased. The path string may include a drive letter. If the path string doesn't specify the root, the current directory path for the drive is prefixed to it. The resulting string may be longer than 64 bytes; that is, this call can delete any directory that *DosMkDir* can create, even if *DosChDir* can't access it. All names in the path must exist and be directories. The last name is the

directory to be deleted. It must be empty; that is, it may contain only the "dot" and "doubledot" entries that were created when it was made.

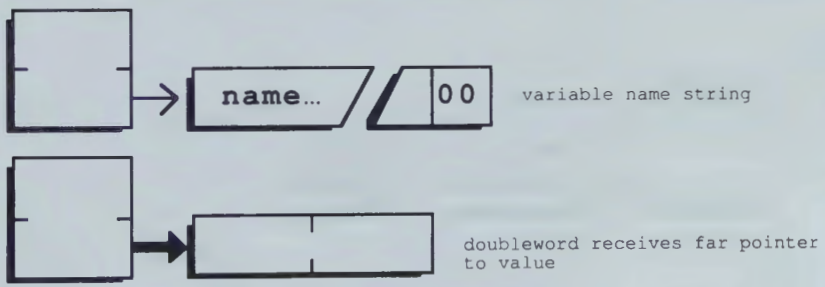
Discussion: *See* 8.6.1, 8.5.3 **Include:** FILES
Related: *DosMkDir*
Family API: available
Errors to consider:

3 Path not found: invalid drive or a member doesn't exist or isn't a directory
5 The last name in the path isn't an empty directory
87 The second parameter must be a doubleword of 0

```
extern unsigned far pascal DOSRMDIR(char far *, long);
#define DosRmDir DOSRMDIR

function DosRmDir       (vars ThePath : char; {e.g. Pathstr[1]}
                         Zero : integer4
                         ) : word; external;
```

DosScanEnv

Search environment for named variable



The environment segment of the running process is searched for a variable with a name that matches the string passed as the first parameter. This name should be spelled in uppercase letters. Its final character should *not* be an equals sign; an equals sign in the environment is a delimiter, not part of a name. If a match is found, the second parameter is updated with the far address of the value string associated with the name. This string may be null; that is, the first character at that address may be the null character. If no match is found, the second parameter is not altered (so it may be preset to the address of a default value string).

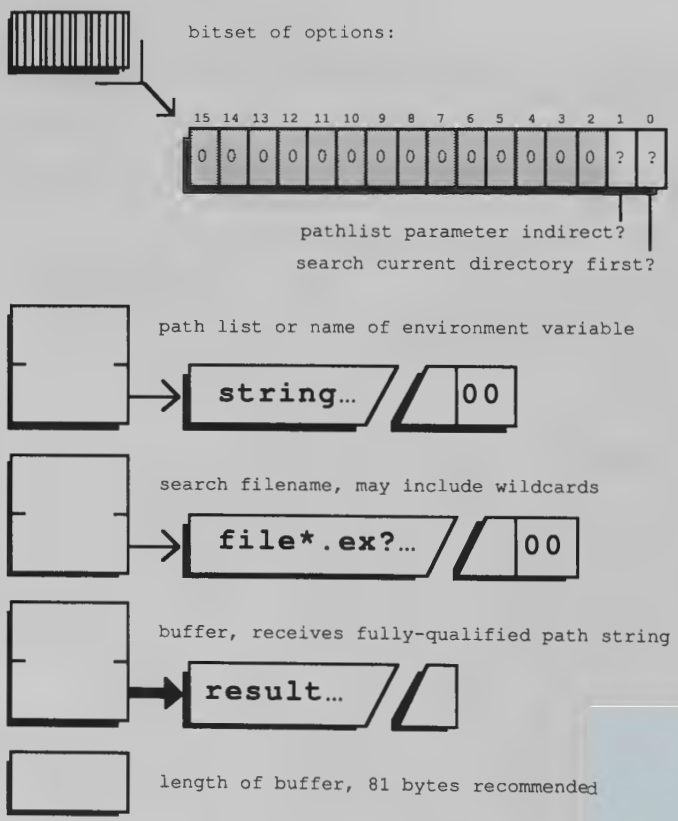
Discussion: *See* 5.1.5.2 **Include:** SYSINFO
Related: *DosGetEnv, DosSearchPath*
Family API: not available

Errors to consider:
203 Name not found in environment (remember: uppercase, no equals-sign)

```
extern unsigned far pascal DOSSCANENV(
                        char far *,     /* name string */
                        char far * far *); /* pointer */
#define DosScanEnv DOSSCANENV

function DosScanEnv     (consts TheName: char; { e.g. Name[1] }
                         vars ToValue: ADSMEM
                         ) : word; external;
```

DosSearchPath

Look for file in list of paths



A list of paths is searched for the first occurrence of a filename. The filename

may contain asterisk and question-mark "wildcards." The fully-qualified path string for the file is returned in a form acceptable to *DosOpen* or *DosSearchFirst*. If option bit 0 is set, the current directory is searched first, before any path in the list. If not, the current directory is not used in the search except as it may turn up in the list of paths.

The list of paths may be given indirectly via the environment. If option bit 1 is set, the second parameter is taken as the name of an environment variable (for instance, "DPATH"). It is looked up in the environment in the manner of *DosScanEnv* and its value there is used as the path list. Otherwise the parameter itself is taken as a path list (one or more path strings separated by semicolons, in the style of the "path" and "dpath" commands).

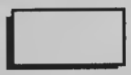
The first match to the filename stops the search. The path on which it was found is copied to the result buffer and the filename is appended to it to form a fully-qualified path string. The filename is copied as it was passed, except that lowercase letters are converted to uppercase and asterisk wildcards are converted to question marks. If no match is found the contents of the buffer are not specified.

Discussion: *See* 5.1.5, 5.1.5.2 **Include:** FILES
Related: *DosOpen, DosFindFirst, DosScanEnv*
Family API: not available
Errors to consider:
2 No matching filename found on any path in the list
203 Variable not found in environment

```
extern unsigned far pascal DOSSEARCHPATH(
                        OPTIONSET,
                        char far *, /* pathlist */
                        char far *, /* filename */
                        char far *, /* buffer */
                        int  /* buffer length */
                        );
#define DosSearchPath DOSSEARCHPATH

function DosSearchPath (Options: OptionSet;
                        consts TheList : char; { e.g. ListStr[1] etc. }
                        consts TheName : char;
                        vars   ThePath : char;
                        TheSize : integer
                        ) : word; external;
```

DosSelectDisk

Set default drive letter



drive number, 1=A, 2=B, 3=C and so forth

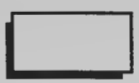
The default disk drive that is assumed whenever a pathname does not contain an explicit drive letter is set. If the disk drive uses demountable volumes (a diskette drive for example), the system may try to read the label of the current volume.

Discussion: *See* 8.1.2, 8.5.2 **Include:** FILES
Related: *DosQCurDisk*
Family API: available
Errors to consider:
15 The drive number must exist in the system

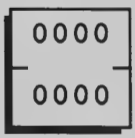
```
extern unsigned far pascal DOSSELECTDISK(int);
#define DosSelectDisk DOSSELECTDISK

function DosSelectDisk (Drive:integer
                        ) : word; external;
```

DosSelectSession

Move related session to foreground



session number returned by DosStartSession



doubleword of zero

The first parameter must be the number of an active session returned when *DosStartSession* created a "related" session. If the caller's session is presently in the foreground it will be moved to the background and the specified session will be moved to the foreground. If it is in the background, nothing happens.

Discussion: *See* 6.2 **Include:** SESSION

Related: *DosSetSession, DosStartSession*
Family API: not available
Errors to consider:
369 The session number must be one returned by *DosStartSession*
371 The session number must be one that is still active

```
extern unsigned far pascal DOSSELECTSESSION(int,long);
#define DosSelectSession DOSSELECTSESSION

function DosSelectSession ( TheSG, AnyZero : integer4
                          ) : word; external;
```

DosSemClear

Release a semaphore

far pointer to local semaphore
or handle of system semaphore

If the semaphore is used as a key, this call gives up ownership gotten with *DosSemRequest*. If it is used as a signal, this call announces that the event has taken place. In either case if any threads were waiting on the semaphore they are made active.If this is a system semaphore created with the exclusive option, only the process that owns it (last set or claimed it) can clear it.

Discussion: *See* 9.4.1 **Include:** ASYNC
Related: *DosSemRequest, DosSemSet, DosSemWait, DosCreateSem*
Family API: not available
Errors to consider:
6 The handle must be one returned by *DosOpenSem* or *DosCreateSem* or be a valid far pointer
101 That's an exclusive semaphore and this process doesn't own it

```
extern unsigned far pascal DOSSEMCLEAR(SEMHANDLE);
#define DosSemClear DOSSEMCLEAR

function DosSemClear    (TheHdle: SemHandle
                         ):word; external;
```

DosSemRequest

Claim ownership of semaphore

far pointer to local semaphore
or handle of system semaphore

maximum delay as a 32-bit
count of milliseconds

If the semaphore is presently clear, it is marked as owned by this thread and control returns at once. If it is owned, this thread is suspended until the semaphore is cleared. At that time all waiting threads will be released; the first to run will gain ownership and the rest will wait again. The delay parameter sets an upper limit on the wait. Use minus-1 (32 binary 1-bits) to allow an indefinite wait; this assures that control will not return until you own the semaphore. Use zero to assure an immediate return even if the semaphore is not clear. When the thread must not be delayed indefinitely, pass an intermediate value and test for error 121 after the call.

If this is not a local but a system semaphore, and if the previous owner terminated without clearing it, error 105 is returned. The meaning of this depends on the design of the program.

Discussion: *See* 9.4.1, 9.4.2 **Include**: ASYNC
Related: *DosOpenSem, DosSemClear*
Family API: not available
Errors to consider:

6 The handle must be one returned by *DosOpenSem* or *DosCreateSem* or a valid far pointer
105 The previous owner terminated without clearing the semaphore
121 The specified time elapsed and the semaphore was still not clear

```
extern unsigned far pascal DOSSEMREQUEST(SEMHANDLE, TIMEINTERVAL);
#define DosSemRequest DOSSEMREQUEST

function DosSemRequest (TheHdle: SemHandle;
                        MaxTime: TimeInterval
                        ): word; external;
```

DosSemSet

Mark a semaphore owned

far pointer to local semaphore
or handle of system semaphore

This call marks a semaphore "owned" by the current thread; as a result a *DosSemWait* or a *DosSemRequest* call will wait. Ordinarily the call is applied to a semaphore used as a signal to put it in the state of "the event has not occurred." It may be applied to a semaphore used as a key to prevent any other thread from claiming it. An attempt to set a system semaphore that was created with the "exclusive" option will fail if the semaphore was not clear.

Discussion: *See* 9.4.1, 9.4.3 **Include:** ASYNC

Related: *DosSemClear, DosSemWait, DosSemRequest, DosCreateSem*

Family API: not available

Errors to consider:

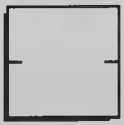
6 The handle must be one returned by *DosOpenSem* or *DosCreateSem* or be a valid far pointer

101 That's an exclusive semaphore and this process doesn't own it

```
extern unsigned far pascal DOSSEMSET(SEMHANDLE);
#define DosSemSet DOSSEMSET

function DosSemSet      (TheHdle: SemHandle
                         ):word; external;
```

DosSemSetWait

Set a semaphore, wait until cleared



far pointer to local semaphore
or handle of system semaphore

maximum delay as a 32-bit
count of milliseconds

If the semaphore is not marked "owned" it is set as by *DosSemSet*; then a wait ensues as for *DosSemWait*. The effect is the same as *DosSemWait* except that a fresh signal is guaranteed; control will not return until the semaphore has been cleared by another thread at least once. The delay parameter sets an upper limit on the wait, as for *DosSemWait*.

Discussion: *See* 9.4.1, 9.4.3 **Include:** ASYNC

Related: *DosSemSet, DosSemWait, DosSemClear*

Family API: not available

Errors to consider:

6 The handle must be one returned by *DosOpenSem* or *DosCreateSem* or be a valid far pointer

101 That's an exclusive semaphore and this process doesn't own it

105 The previous owner terminated without clearing the semaphore

121 The specified time elapsed and the semaphore was still not clear

```
extern unsigned far pascal DOSSEMSETWAIT(SEMHANDLE, TIMEINTERVAL);
#define DosSemSetWait DOSSEMSETWAIT

function DosSemSetWait (TheHdle: SemHandle;
                        MaxTime: TimeInterval
                        ): word; external;
```

DosSemWait

Wait until semaphore cleared

far pointer to local semaphore
or handle of system semaphore

maximum delay as a 32-bit
count of milliseconds

If the semaphore is clear control returns at once. If not, this thread is suspended until the semaphore is cleared. At that time this and any other waiting threads will be released to run again. When its turn comes, this thread will repeat its test and return if the semaphore is clear. (It is possible for another thread to run first and set the semaphore; then this thread will wait again.) The delay parameter sets an upper limit on the wait. Use minus-1 (32 binary 1-bits) to allow an indefinite wait; this assures that control will not return until the semaphore has remained clear long enough for this thread to be dispatched. Pass zero to assure an immediate return even if the semaphore is not clear. When the thread must not be delayed indefinitely, pass an intermediate value and test for error 121 after the call. If this is not a local but a system semaphore, and if a different process had set the semaphore and terminated without clearing it, error 105 is returned. The meaning of this depends on the design of the program.

Discussion: *See* 9.4.1, 9.4.3 **Include:** ASYNC

Related: *DosSemSet, DosSemClear, DosOpenSem*

Family API: not available

Errors to consider:

6 The handle must be one returned by *DosOpenSem* or *DosCreateSem* or be a valid far pointer

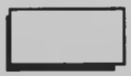
105 The previous owner terminated without clearing the semaphore

121 The specified time elapsed and the semaphore was still not clear

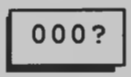
```
extern unsigned far pascal DOSSEMWAIT(SEMHANDLE, TIMEINTERVAL);
#define DosSemWait DOSSEMWAIT

function DosSemWait     (TheHdle: SemHandle;
                         MaxTime: TimeInterval
                         ): word; external;
```

DosSendSignal

Send break signal



process ID of target



signal: 1=control-C, 4=control-break

If the specified process exists and is a child of the calling process, the type of signal specified by the second parameter is delivered to it or one of its child processes. The signal is given to the most remote descendant that has not blocked that type of signal, or to the process itself if it has no children or all are blocking the signal. The signal will be received by the signal handler for the type of signal. The default signal handler will convert it into a termination signal, or the process may have installed a handler to ignore the signal.

Discussion: *See* 5.3.1 **Include:** PROCESS
Related: *DosFlagProcess, DosHoldSignal, DosSetSigHandler*
Family API: not available
Errors to consider:
156 The specified process has blocked these signals
162 A signal of this type is already pending at that process
186 The third parameter must be 1 or 4
303 The first parameter must be an existing process
305 The process must be a child of this one

```
extern unsigned far pascal DOSSENDSIGNAL(int,int);
#define DosSendSignal DOSSENDSIGNAL

function DosSendSignal(ToWhom: ProcessID;
                       WhichOne: integer
                        ):word; external;
```

DosSetCP

Set code page for ASCII collating sequence

code page number

must be zero

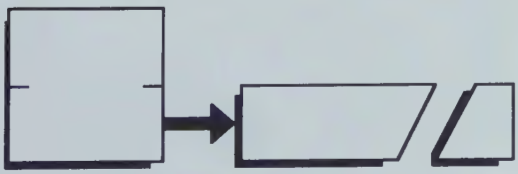
The specified code page (see Table 5.3, p. 162) is set as the default code page for *DosGetCollate* and *DosCaseMap*, and perhaps for printer output.

Discussion: *See* 5.2.3 **Include:** SYSINFO
Related: *KbdSetCP*, *VioSetCP*
Family API: available
Errors to consider:
398 The code page must be one that is supported

```
extern unsigned far pascal DOSSETCP(int, int);
#define DosSetCP DOSSETCP

function DosSetCP       (Codepage : integer;
                         AnyZero : integer
                         ) : word; external;
```

DosSetDateTime

Update date and time information



date and time Structure, see Figure 5.1, p.143

Updated information on the time and date is stored into the system global information segment. All fields of the date and time structure except the day-of-week number are updated and so must be correct. The best way to use this call is to use *DosGetDateTime* to fill the structure, modify only the fields to be

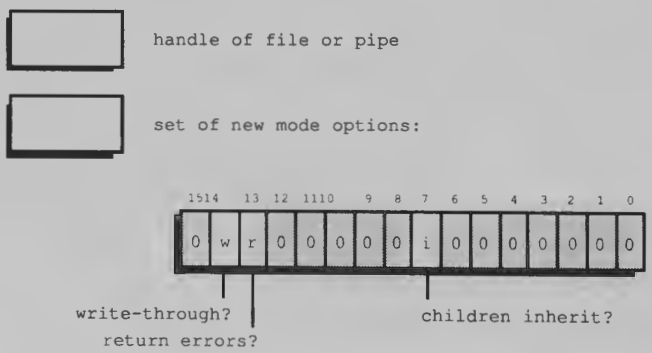
changed, and pass the same structure to this call. The day-of-week number is computed by the system from the new date information. When changing the hour for daylight savings time, remember to change the time-zone value by 60 minutes as well.

Discussion: *See* 5.1.2 **Include:** SYSINFO
Related: *DosGetDateTime*
Family API: available
Errors to consider:
327 At least one field is inconsistent or out of its proper range

```
extern unsigned far pascal DOSSETDATETIME(struct DATETIME far *);
#define DosSetDateTime DOSSETDATETIME

function DosSetDateTime(consts TheInfo: DateTime) : word;
                  external;
```

DosSetFHandState

Set handle options



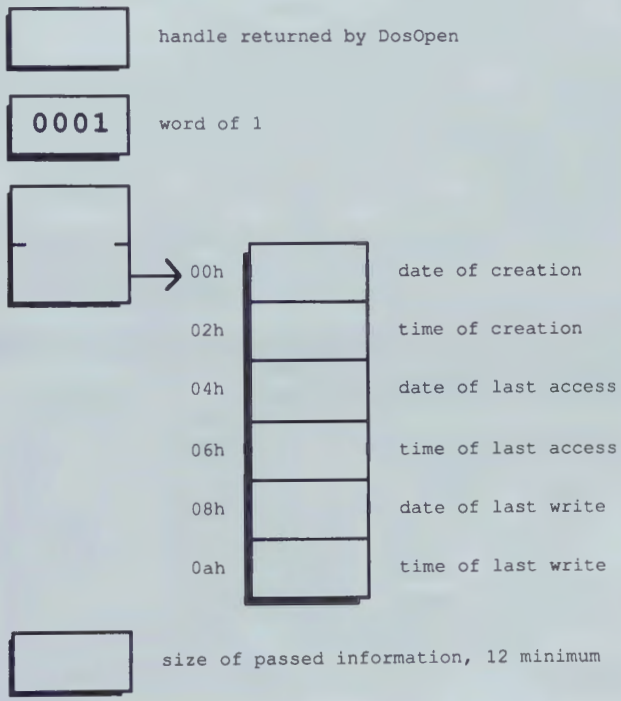
Three of the options that are originally set by the Mode word input to *DosOpen* are changed. Only the bits for the Write-through, Return-errors, and Inherit options may be nonzero; all other bits in the word must be zero.

Discussion: *See* 8.10.1, 8.7.3 **Include:** FILES
Related: *DosQFHandState, DosOpen*
Family API: Handle isn't checked; all option bits must be zero.
Errors to consider:
6 The handle must be one returned by *DosOpen* and not yet closed
87 Only the Write-through, Return-errors, and Inheritance bits may be nonzero

```
extern unsigned far pascal DOSSETFHANDSTATE(HANDLE,OPENMODE);
#define DosSetFHandState DOSSETFHANDSTATE

function DosSetFHandState(TheHandle:Handle; NewMode:OpenMode
                        ) : word; external;
```

DosSetFileInfo

Set file timestamps



The directory timestamp information for the specified file is set. The file must have been opened with an access mode of write-only or read-write. If a date/time word pair contain binary zero, that timestamp is left unchanged. Nonzero in either a date or time cause that pair to be updated. Only timestamps that are maintained by the system (only the last-write fields, in version 1) may be updated; others must be zero. No consistency checks are applied to the passed values.

Discussion: *See* 8.6.5.3 **Include:** FILES

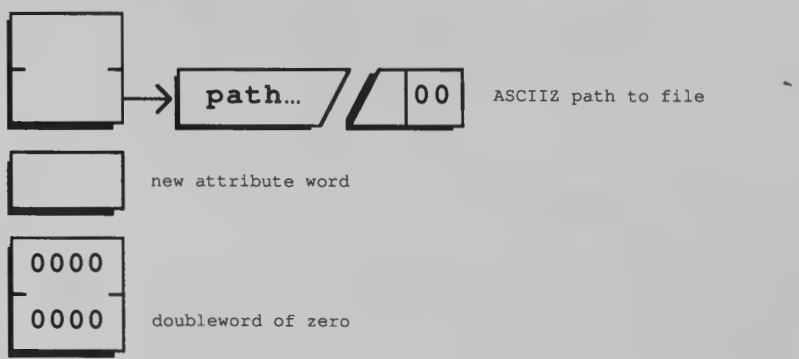
Related: *DosQFileInfo, DosOpen, DosFindFirst*
Family API: available
Errors to consider:

5 The file must be open for write access
6 The handle must be open to a disk file
87 Only maintained fields may be nonzero
122 The size parameter must be at least 12
124 The second parameter must be 1

```
extern unsigned far pascal DOSSETFILEINFO(
                        HANDLE,
                        int, /* must be 1 */
                        struct FileDates far *,
                        int); /* sizeof(struct FileDates) */
#define DosSetFileInfo DOSSETFILEINFO

function DosSetFileInfo(     TheHandle : Handle;
                             TheLevel : integer; { must be 1 }
                        vars TheDates : FileDates;
                             TheSize  : integer {sizeof(FileDates)}
                        ) : word; external;
```

DosSetFileMode

Set file attribute word



The attributes of the specified file are set to new values. The Directory attribute cannot be changed, only the read-only, hidden, system and archive attributes.

Discussion: *See* 8.3, 8.6.4 **Include:** FILES
Related: *DosQFileMode, DosQFileInfo*

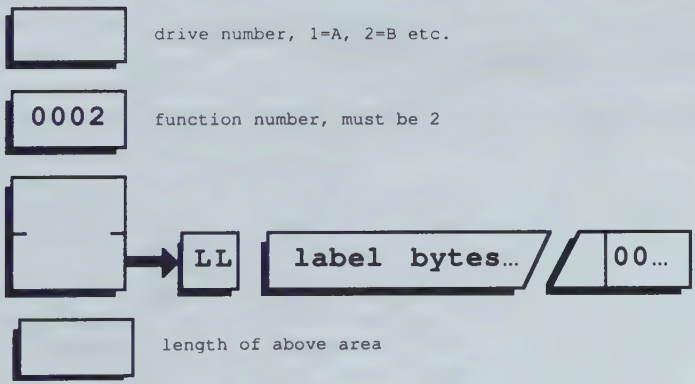
Family API: available
Errors to consider:

2 File not found (but path is OK)
3 Path not found: invalid drive or a member doesn't exist or isn't a directory
5 The target file is a directory
87 Only supported attributes may be nonzero in the attribute word

```
extern unsigned far pascal DOSSETFILEMODE(
                        char far *, /* path */
                        ATTRWORD,
                        long        /* zero */
                        );
#define DosSetFileMode DOSSETFILEMODE

function DosSetFileMode(vars ThePath : char;
                             TheAttr : AttrWord;
                             AnyZero : integer4
                        ) : word; external;
```

DosSetFSInfo

Set disk volume label



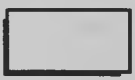
The volume label on the disk in the specified drive is written. The label text is supplied as an ASCIIZ string, prefixed by a byte containing the length of the string data (exclusive of the null). The length parameter following the label parameter may be greater than the size of the string. The volume label string may contain only characters that are permitted in a filename. Lowercase letters will be converted to uppercase.

Discussion: *See* 8.5.5 **Include:** FILES
Related: *DosQFSInfo*
Family API: available
Errors to consider:
15 The drive number must exist in the system
122 The length parameter is less than size of string
123 The label string contains unpermitted characters
124 The function number must be 2
154 The label string may not have more than 11 bytes of text

```
extern unsigned far pascal DOSSETFSINFO(
                        int, /* drive */
                        int, /* 2 */
                        struct FSVolLabel far *,
                        int /* sizeof(label) */
                        );
#define DosSetFSInfo DOSSETFSINFO

function DosSetFSInfo  (     Drive : integer;
                             Funcno: integer;
                        vars LabelText : FSVolLabel;
                             LabLen : integer
                        ) : word; external;
```

DosSetMaxFH

Set limit on file handles for a process



number of file handles

A new limit is set on the maximum number of files that the current process may have open simultaneously. The initial limit is 20, of which at least six are initially reserved by OS/2, so that 14 files may be opened by any process. The new limit may not be set above 255, nor below the number of handles currently in use.

Discussion: *See* 8.4 **Include:** FILES
Family API: not available
Errors to consider:
87 The parameter must be no less than the existing number of handles and no more than 255

```
extern unsigned far pascal DOSSETMAXFH( int );
#define DosSetMaxFH DOSSETMAXFH

function DosSetMaxFH    ( MaxFH : integer
```

DosSetPrty

Set priority of thread or process

000?	option: 0=apply to all threads of a process, 1=apply to all threads of process and children, 2=apply to single specified thread
000?	class: 0=no change to class, 1=idle-time class, 2=regular class, 3=time-critical class
xx??	change in priority, -31 to zero to +31
	process ID (option zero or 1) or thread ID (option 2)

The priority and/or dispatching class of one or more threads is changed. A priority ranges from 0 to 31, with 0 being the usual and default value and higher numbers receiving better service. Priority operates within one of three classes. The idle-time class defers to all other classes; regular class (the usual one) defers to time-critical. The first parameter chooses the scope of the change. If it is 2, the change applies only to the thread specified by the fourth parameter, which may be zero to indicate the current thread. Otherwise the fourth parameter must be a process ID (zero means the current process), and the change is applied to all threads in that process, and to all its descendant processes as well if the option is 1. The second parameter specifies a dispatching class, with zero indicating no change. If the class is changed, the priority is reset to zero before the third parameter is examined. The third parameter specifies a change in priority, with zero making no change, negative numbers reducing priority, and positive numbers raising it. The parameter is treated as a byte and sign-extended to a word, so 254 (0feh) is treated as -2 (fffeh). Values that would take the priority outside the limit of zero or 31 take it only to the limit and no further.

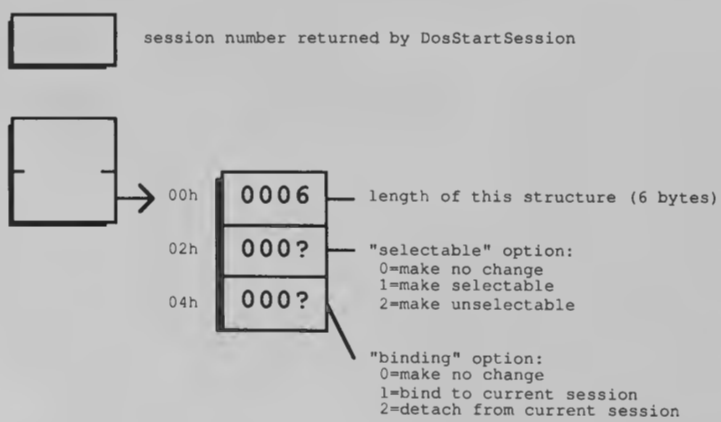
Discussion: *See* 9.1.5, 9.1.5.3 **Include:** ASYNC
Related: *DosGetPrty*
Family API: not available
Errors to consider:

303 The process specified must exist
304 The priority change must lie between -31 and +31 inclusive
305 Only a process that is a child of the current process may be changed
307 The class must be 0, 1, 2 or 3
308 The option must be 0 1 or 2
309 The thread specified must exist in the current process

```
extern unsigned far pascal DOSSETPRTY(
                        int, /* scope */
                        int, /* class */
                        int, /* change in priority */
                        THREADID);
#define DosSetPrty DOSSETPRTY

function DosSetPrty     ( TheScope : integer;
                          TheClass : integer;
                          PrtyChange : integer;
                          TheObject : ThreadID
                        ) : word; external;
```

DosSetSession

Change association of parent and child sessions



The first parameter must be the number of an active session, returned when *DosStartSession* created a "related" session. The session is altered in one or both of two important ways, depending on the options passed. If the second field is 2, the session is made unselectable; that is, the operator will not be able to bring it

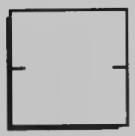
to the foreground. It will not come up when cycling through the screen groups with Alt-Escape. It will still appear in the Session Manager's "switch to a running program" menu, but if the operator selects it, an error message will pop up. If the second field is 1, the session is made selectable (the normal condition). If the third field is 1, the session is bound to the caller's session such that whenever the operator selects the caller's session (either by Alt-Escape or from the Session Manager's menu), the specified session will appear instead. The current session will become invisible and will have no access to keyboard, screen, or mouse until the binding is undone. If the specified session is selectable, it will come up in its own turn as well as that of the current session. The binding can be undone in three ways: by using this call with a 2 in the third field; by using this call to bind to a different session; or by the termination of the bound session.

Discussion: *See* 6.2 **Include:** SESSION
Related: *DosStartSession, DosStopSession*
Family API: not available
Errors to consider:
24 The structure length must be 6
369 The session number must be one returned by *DosStartSession*
371 The session number must be one that is still active
455 The "Binding" option must be zero, 1 or 2
456 The "Selectable" option must be zero, 1 or 2

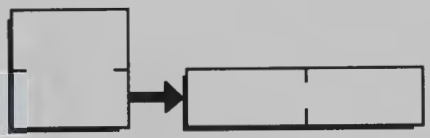
```
struct STATUSDATA {
        int std_size; /* sizeof(struct SESSIONSTATUS) */
        int std_select, std_bind;
                        };
extern unsigned far pascal DOSSETSESSION(
                                int, /* session id */
                                struct STATUSDATA far *);
#define DosSetSession DOSSETSESSION

type
        StatusData = record
                stdSize, { 6 }
                stdSelect,
                stdBind : integer
        end;
function DosSetSession ( TheSession : integer;
                        consts TheParms : StatusData
                        ) : word; external;
```

DosSetSigHandler

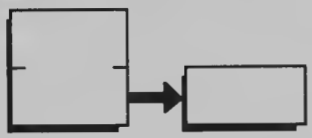
Set handler for external signals



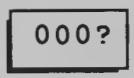
when action = 2, far pointer to entry point
of handler routine, otherwise 32-bit zero



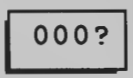
doubleword receives address of
entry point to previous handler



word receives previous action (0, 1, 2, or 3)



action:
0=set system default handler
1=ignore this signal type
2=invoke specified entry point on signal
3=return an error to user of DosFlagProcess
4=acknowledge last signal of this type



signal number at issue:
1=control-C
3=process termination for any cause
4=control-break
5,6,7=flag signals A,B,C

One of a variety of actions is performed regarding the signal number given as the last parameter, depending on the action number in the next to last parameter. In all cases something is stored in the second and third parameters. When the action is zero, 1, 2 or 3, the stored values may be used at a later time to reinstate the prior signal handling method.

When the action is zero, the default system handler for the signal is established. The default handlers are in place when a process begins. The handlers for signals 1 and 3 issue a signal 3; that for signal 3 terminates the process; those for signals 5, 6 and 7 acknowledge the signal but take no other action.

When the action is 1, the signal is ignored, which means to acknowledge it (so

that it does not remain pending) but to do nothing else. If the Control-C and Control-Break signals are ignored, the process cannot be terminated by them. If the termination signal is ignored, the process cannot be killed by external events and will run until it calls *DosExit* or causes a hardware trap.

When the action is 3, signals of the specified type will be rejected with an error code to the signaller. This is ordinarily done for signals 5, 6 and 7. Rejection of Control-C and Control-Break is the same as ignoring them. Rejection of termination causes an error code return to the user of *DosKillProcess*.

When the action is 2, the entry point addressed by the first parameter is set as the handler, which means it will be invoked when a signal of this type arrives. The handler will be entered as a far subroutine call under the first thread of the process, with a single word parameter on the stack (zero for signals 1, 3 and 4; the argument of *DosFlagProcess* for signals 5, 6 and 7). When the handler returns, the first thread resumes at the point of interruption (however, unless the handler preserves all registers and the flags, the first thread will perceive a random perturbation of its registers). A second signal of the same type remains pending until the handler acknowledges the current one; if it never does, third and later signals will be rejected. Action 4 is used within a signal handler to acknowledge a signal. If a second signal is pending, the handler routine may be invoked immediately upon completion of this call, possibly entering the handler before it exits from the first signal.

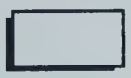
Discussion: *See* 5.3.1 **Include:** PROCESS
Related: *DosHoldSignal, DosFlagProcess, DosKillProcess, DosSendSignal*
Family API: Only actions 0, 1 and 2 may be done, and only for signals 1 and 3.
Errors to consider:
1 The action must be in 0-4 and the signal number in 1-7
209 The signal number must be 1, 3, 4, 5, 6 or 7

```
extern unsigned far pascal DOSSETSIGHANDLER(
                        ADSPROC,         /* far * new hdlr */
                        ADSPROC far *,   /* ->far * old hdlr */
                        int far *,       /* ->old action */
                        int,             /* action */
                        int);            /* signal */
#define DosSetSigHandler DOSSETSIGHANDLER

function DosSetSigHandler(
                        NewHandler : ADSPROC; { use NilADSPROC for nil }
                    vars OldHandler : ADSPROC; { receives old hdl addr }
                    vars OldAction : integer;  { SigActions but 16 bits }
                        NewAction : SigActions;
```

```
TheSignal : SigTypes
        ): word; external;
```

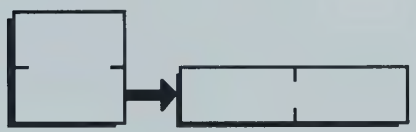
DosSetVec

Set handler for hardware interrupt



```
interrupt number:
 0=divide by zero
 4=INTO instruction
 5=BOUND instruction
 6=invalid opcode
 7=no Numeric Processor
16=Numeric Processor error
```

far pointer to entry point of handler code



receives far pointer to prior handler

A handler is set up to intercept one of six hardware traps. When a hardware trap occurs, the current thread will be diverted to the handler address. The stack will reflect a hardware interrupt; that is, the machine flag word will be first on the stack followed by a far return address. If the handler is written in assembly language it may exit with an IRET instruction, reloading the flags register. Â handler in a high level language could treat the flags word as a word parameter.

The address of the prior handler is returned; this value can be used later to reinstate the prior handler. So long as a handler is active for interrupt 7, the Machine Status Word used within the current process will be set to indicate that no Numeric Processor is installed, regardless of the true hardware configuration. After any interrupt but 16, return to the point of interruption will re-execute the instruction that caused the interrupt (interrupt 16 may occur some time after the Numeric Processor instruction that provoked it). A handler for interrupt 7 will generally examine and emulate the Numeric Processor instruction at the return address, advance the return address over it, and return. Handlers for other interrupts will generally either terminate the process with *DosExit,* or alter the return address so as to "return" to an error routine.

Discussion: *See* 5.3.2 **Include:** PROCESS
Related: none
Family API: available
Errors to consider:
1 The interrupt number must be 0, 4, 5, 6, 7 or 16

```
extern unsigned far pascal DOSSETVEC(
                        int,            /* 0, 4, 5, 6, 7, 16 */
                        ADSPROC,        /* far * new hdlr */
                        ADSPROC far *); /* ->far * old hdlr */
#define DosSetVec DOSSETVEC

function DosSetVec      (TheIntNum : integer; { an Int.. const }
                         NewHandler : ADSPROC;
                    vars OldHandler : ADSPROC
                         ) : word; external;
```

DosSetVerify

Set verify-write switch on or off

000?	word of 0 for on, or 1 for off

The state of the write-verify switch for this process is set to on or off. Disk writes initiated by this process will be verified by reading back, or not, accordingly. When write-verify is on, an error on reading back will only be returned to the program if the file handle is in return-errors mode.

Discussion: *See* 8.5.1, 8.7.3.4 **Include:** FILES
Related: *DosQVerify, DosOpen*
Family API: available
Errors to consider:
87 The parameter must be 0 or 1

```
extern unsigned far pascal DOSSETVERIFY(int);
#define DosSetVerify DOSSETVERIFY

function DosSetVerify  (Switch:integer
                        ):word; external;
```

DosSleep

Suspend thread for an interval

32-bit integer count of milliseconds

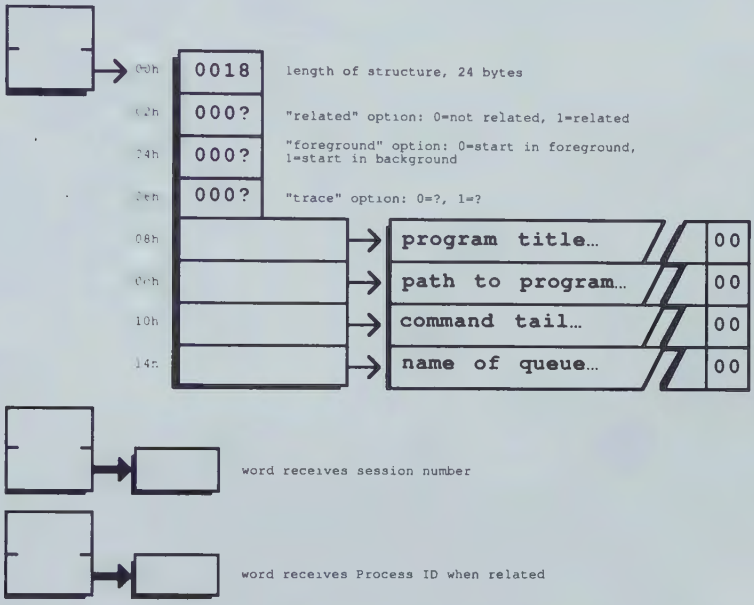
This thread is suspended for an interval at least as long as the number of milliseconds given in the parameter. The thread is always suspended while every other eligible thread is given one chance to run; hence this call with a parameter of zero has the effect of temporarily handing off the use of the CPU to any waiting thread and then regaining control. The smallest resolvable time interval is available via *DosGetInfoSeg*, and is typically 31 milliseconds. The actual interval before control returns is not predictable, since it depends on the other activity in the system and on the relative priority of this thread. The additional time over the specified interval could be near zero milliseconds, or it could range as high as the maximum-wait interval (also available via *DosGetInfoSeg*).

Discussion: *See* 9.5, 9.5.2 **Include:** ASYNC
Related: *DosTimerAsync*
Family API: available
Errors to consider: none

```
extern void far pascal DOSSLEEP(TIMEINTERVAL);
#define DosSleep DOSSLEEP

procedure DosSleep (HowLong:TimeInterval);
                        external;
```

DosStartSession

Create application session



The data structure passed as the first parameter is examined. The options in the second, third and fourth fields must be either zero or 1. The sixth field must be the address of an ASCIIZ character string that is the fully-qualified pathname of an executable program. If that program file can be found, a new screen group is created and the program is loaded and started as a process in that new screen group. The combination of process and screen group is a session. The ASCIIZ string addressed by the fifth field will be displayed as the title of the new screen group in the Session Manager's "switch to a running program" menu. The new process receives an environment segment that contains no named variables at all. It receives two argument strings, the strings addressed by the sixth and seventh fields of the structure.

If the "foreground" option is zero and the screen group of the calling process is in the foreground, the new screen group will be switched to the foreground, as if the operator had selected it. Otherwise the new screen group will be in the background, but the operator may select it with the usual hot keys. If the "related" option is zero, the new session is independent, and the calling process has no further control over it. Nothing is returned in the second and third

parameter words. The session exists until the new process and its children end, then it terminates.

When the "related" option is 1, the new session can be controlled in certain ways. A session number (an arbitrary number, not the same as the screen group ID that can be found in the data returned by *DosGetInfoSeg*) is returned in the second parameter; this can be used with *DosSelectSession*, *DosSetSession*, and *DosStopSession*. The process ID of the new process is returned in the third parameter; this can be used with *DosKillProcess*, and *DosFlagProcess* If a queue-name string is specified in the last field when a related session is started, a notification item will be written to that queue when the session ends. The far pointer in the queue item will address two words: the session number and the exit code of the child process, in that order.

Discussion: *See* 6.2 **Include:** SESSION

Related: *DosSetSession, DosSelectSession, DosStopSession, DosCreateQueue, DosReadQueue*

Family API: not available

Errors to consider:

DosOpen error codes 2, 3, 4, 5, 32 and 123 may be returned

24 The first word of the structure must be its length, 24 decimal

203 The program file couldn't be found, specify a drive and pathname

370 The system limit on screen groups has been exceeded

453 The "foreground" option must be zero or 1

454 The "related" option must be zero or 1

478 The "trace" option must be zero or 1

```
extern unsigned far pascal DOSSTARTSESSION(
                                struct STARTDATA far *,
                                int far *,
                                PROCESSID far *
                                );
#define DosStartSession DOSSTARTSESSION

function DosStartSession(consts TheParms : StartData;
                         vars TheSess : integer;
                         vars RelPid  : ProcessID
                        ) : word; external;
```

DosStopSession

End child session

The option word appears to have no effect whatever in the first release of the system. Code it as zero in case a later system begins to test it. The session number in the second parameter is examined; it must be the number returned by *DosStartSession* when it created a "related" session. If the session is bound to the current process (see *DosSetSession)* the binding is severed. Then a termination signal (see *DosKillProcess*) is sent to the process that was created by *DosStartSession* and to all its descendants. When the last one of that family of processes has terminated, the screen group is also terminated and becomes available for creating a new session. If a queue name was specified to *DosStartSession*, a queue item is written to it when the screen group terminates.

Discussion: *See* 6.2 **Include:** SESSION

Related: *DosStartSession, DosSetSession*

Family API: not available

Errors to consider:

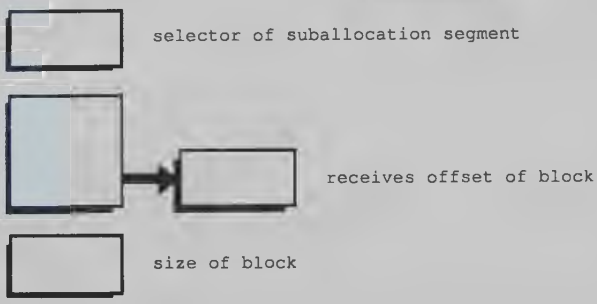
369 The session number must be one returned by *DosStartSession*

371 The session number must be one that is still active

```
extern unsigned far pascal DOSSTOPSESSION(int, int, long);
#define DosStopSession DOSSTOPSESSION

function DosStopSession(TheOption, TheSession, AnyZero: integer4
                        ) : word; external;
```

DosSubAlloc

Suballocate storage



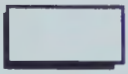
The referenced segment must have been prepared by *DosSubSet*. A block of the requested size is allocated within it and the offset to the block is returned. The suballocation pointers are updated with interrupts disabled, so blocks may be allocated and freed from multiple asynchronous processes.

Discussion: *See* 7.3.5 **Include:** STORAGE
Related: *DosSubFree, DosSubSet*
Family API: available
Errors to consider:
311 Not enough space remains in the segment
313 The size parameter must be nonzero and less than the segment size

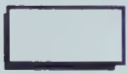
```
extern unsigned far pascal DOSSUBALLOC(
                        SELECTOR,
                        unsigned far *,
                        unsigned);
#define DosSubAlloc DOSSUBALLOC

function DosSubAlloc    (     TheSel : Selector; { typically ADS.S }
                        vars Offset : word;      { typically ADS.R }
                             TheSize: word
                        ) : word; external;
```

DosSubFree

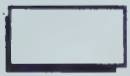
Release suballocated storage



selector of suballocation segment



offset of block being released



size of released block

The offset and size must be those returned by a prior call to *DosSubAlloc*. The block is returned to the suballocation pool for reuse. The suballocation pointers are updated with interrupts disabled, so blocks may be allocated and freed from multiple asynchronous processes.

Discussion: *See* 7.3.5 **Include:** STORAGE

Related: *DosSubAlloc, DosSubSet*

Family API: available

Errors to consider:

312 The block described overlaps on other blocks

313 The size word must be nonzero and smaller than the size of the segment

```
extern unsigned far pascal DOSSUBFREE(
                        SELECTOR,
                        unsigned,
                        unsigned);
#define DosSubFree DOSSUBFREE

function DosSubFree     (TheSel : Selector; { typically ADS.S }
                         Offset : word;     { typically ADS.R }
                         TheSize : word
                         ) : word; external;
```

DosSubSet

Initialize a segment for suballocation

selector returned by allocation

0001 when initialing a segment,
0000 when expanding it

current size of segment, zero meaning 65,536

The referenced segment must have been allocated by *DosAllocSeg* or *DosAllocShrSeg*. When the option word is 1 for initializing, this call prepares the entire segment to be a suballocation pool. This must be done before *DosSubAlloc* is used with this segment. If it is repeated later, the pool is reinitialized and all prior suballocations are lost. When the option word is 0 for expanding, any space added to the segment by *DosReallocSeg* is incorporated into the suballocation pool. This is the only way to make added space available through *DosSubAlloc*. Once suballocation has begun, the segment should not be made smaller. *DosReallocSeg* shrinks a segment by truncating it, which destroys the internal pointers set up during suballocation. If it is necessary to reduce the size of a suballocated segment, the only course is to reinitialize it completely using option 1 of this call. A suballocated segment may be shared between processes using any of the three methods (*DosGetShrSeg*, *DosGetSeg*, *DosGiveSeg*). The *DosSubAlloc* and *DosSubFree* calls may be used simultaneously from asynchronous processes.

Discussion: *See* 7.3.5 **Include**: STORAGE

Related: *DosAllocSeg, DosReallocSeg, DosSubAlloc, DosSubFree*

Family API: available

Errors to consider:

310 With option 0 the size may not be less than the current size

313 The size parameter may not fall between 1 and 11 bytes

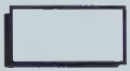
314 The option must be 0 or 1

```
extern unsigned far pascal DOSSUBSET(
                        SELECTOR,
                        unsigned,
                        unsigned);
#define DosSubSet DOSSUBSET

function DosSubSet      (TheSel : Selector;
```

```
Option : word;
TheSize: word
) : word; external;
```

DosSuspendThread

Force thread into suspension



thread ID of thread to be suspended

The specified thread must be part of the current process. If it is presently executing in OS/2 system code it may remain active until it exits the system code. If, or when, it is executing in application code it will be suspended and will not be dispatched again until released with *DosResumeThread.*

Discussion: *See* 9.1.3 **Include:** ASYNC

Related: *DosResumeThread, DosCreateThread, DosExit*

Family API: not available

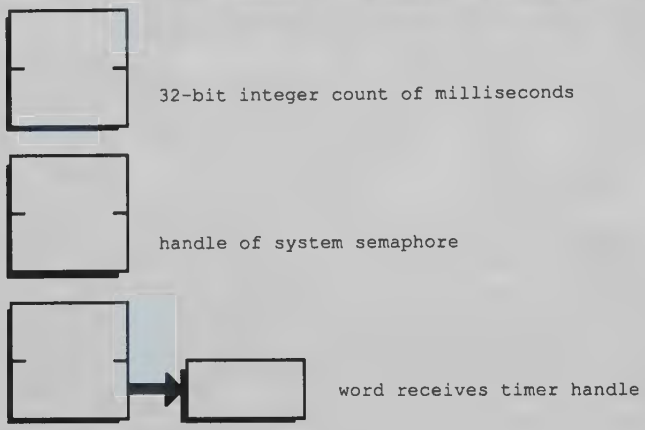
Errors to consider:

309 The thread must be one that exists in this process

```
extern unsigned far pascal DOSSUSPENDTHREAD( THREADID );
#define DosSuspendThread DOSSUSPENDTHREAD

function DosSuspendThread( TheThrd : ThreadID
                         ) : word; external;
```

DosTimerAsync

Start interval timer



The system semaphore named by the second parameter should be set (using *DosSemSet*) before this call. After a delay at least as long as specified by the first parameter, that semaphore will be cleared as by *DosSemClear*, awakening any threads that have suspended on it with *DosSemWait*. The third parameter receives a handle that may be used to cancel the timer before its expiration using *DosTimerStop*. The smallest resolvable time interval is available via *DosGetInfoSeg*. The actual interval before the semaphore is cleared is not predictable, since it depends on the other activity in the system, but will typically exceed the specified interval by no more than a few times the minimum timeslice interval (also available via *DosGetInfoSeg*).

Discussion: *See* 9.5.3 **Include:** ASYNC

Related: *DosTimerStart, DosTimerStop, DosSleep*

Family API: not available

Errors to consider:

91 System timer space exceeded

323 Second parameter must be a handle returned by *DosCreateSem* or *DosOpenSem*

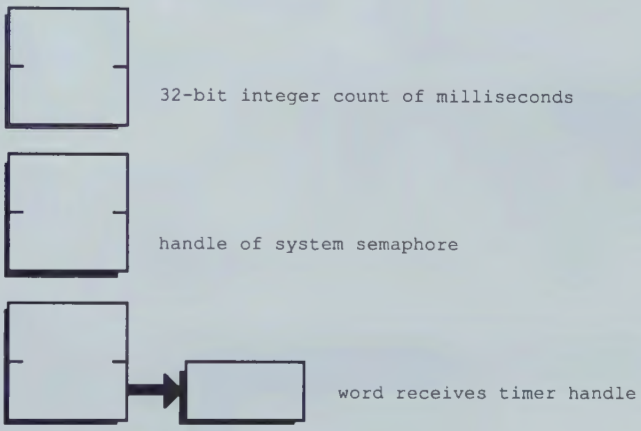
324 System timer space exceeded

```
extern unsigned far pascal DOSTIMERASYNC(
                        TIMEINTERVAL,
                        SEMHANDLE,
                        HANDLE far *);
```

```
#define DosTimerAsync DOSTIMERASYNC

function DosTimerAsync (HowLong: TimeInterval;
                        WhatSem: SemHandle;
                        vars TheHdl : Handle
                        ) : word; external;
```

DosTimerStart

Start periodic timer signal



The system semaphore named by the second parameter will be cleared (as by *DosSemClear*) at regular intervals approximately equal to the first parameter. The system never sets the semaphore, only clears it. Any thread may wait for the next interval to end by setting the semaphore and waiting on it with *DosSemWait*. The third parameter receives a handle that may be used to cancel the timer using *DosTimerStop*. The smallest resolvable time interval is available via *DosGetInfoSeg*. There is no promise that a waiting thread will be awakened and dispatched in time to catch the next interval. An awakened thread may be delayed by as much as a few times the minimum timeslice (available via *DosGetInfoSeg*); the nearer the periodic interval approaches the timeslice interval the more likely it is that threads will not be dispatched at the timer rate.

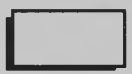
Discussion: *See* 9.5.3 **Include:** ASYNC
Related: *DosTimerAsync, DosTimerStop*
Family API: not available
Errors to consider:
91 System timer space exceeded

323 Second parameter must be a handle returned by *DosCreateSem* or *DosOpenSem*
324 System timer space exceeded

```
extern unsigned far pascal DOSTIMERSTART(
                        TIMEINTERVAL,
                        SEMHANDLE,
                        HANDLE far *);
#define DosTimerStart DOSTIMERSTART

function DosTimerStart (HowLong: TimeInterval;
                        WhatSem: SemHandle;
                        vars TheHdl : Handle
                        ): word; external;
```

DosTimerStop

Cancel periodic or interval timer



handle returned by DosTimerAsync or DosTimerStart

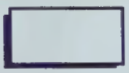
The specified timer is cancelled. Since timers operate asynchronously, it is impossible to be certain of cancelling a timer before its interval expires and the associated semaphore is cleared. Even if the semaphore is not clear when this call is made, it might be cleared during it. Once this call has been completed, the semaphore will not be cleared again.

Discussion: *See* 9.5.3 **Include:** ASYNC
Related: *DosTimerAsync, DosTimerStart*
Family API: not available
Errors to consider:
326 the handle must be one returned by *DosTimerStart* or *DosTimerAsync*

```
extern unsigned far pascal DOSTIMERSTOP(HANDLE);
#define DosTimerStop DOSTIMERSTOP

function DosTimerStop  (TheHdl:Handle) : word; external;
```

DosUnlockSeg

Make storage discardable



selector returned at allocation

The referenced segment must have been allocated via *DosAllocSeg* with the disposable option. If its lock reference count is greater than zero, it is decremented. If that brings the count to zero, the segment is made discardable; if the system needs storage space for allocation, the segment may be discarded. The segment may remain an accessible part of the address space for an unpredictable length of time after this call, depending on the actions of other processes that share it and on the demand for storage system-wide. Since it could become inaccessible at any instant, it should be assumed to be unusable from the moment the call is made. If its contents are needed, use *DosLockSeg* to recover it.

Discussion: *See* 7.3.7 **Include:** STORAGE

Related: *DosLockSeg, DosAllocSeg*

Family API: not available

Errors to consider:

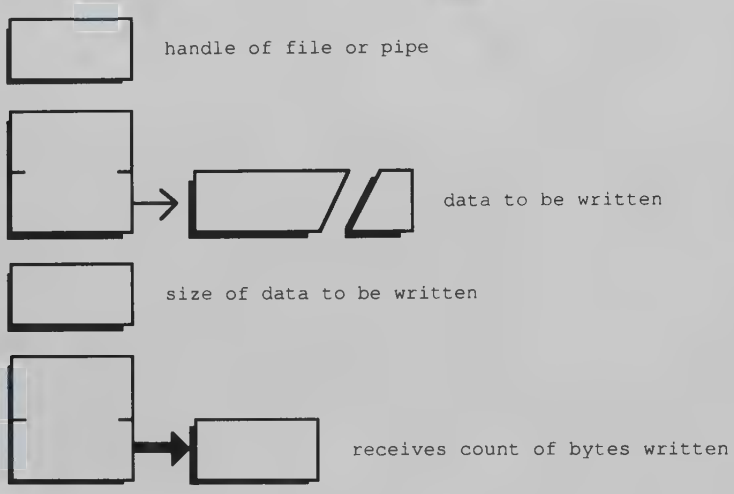
5 The specified segment doesn't exist or wasn't allocated with the right option

158 The segment wasn't locked (its reference count is zero)

```
extern unsigned far pascal DOSUNLOCKSEG(SELECTOR);
#define DosUnlockSeg DOSUNLOCKSEG

function DosUnlockSeg  (TheSel:Selector
                        ) : word; external;
```

DosWrite

Write data to file or pipe



The requested number of bytes are written, if possible. A buffer size of zero is not an error; no bytes are written. The count of bytes written is returned; it will usually be the same as the size of the buffer; however it should be checked. If the count is different some device limit prevented the output, for example there was not sufficient disk space. The file read/write pointer will be incremented by the value of the returned count.

Discussion: *See* 8.11.3, 8.11.4 **Include:** FILES
Related: *DosOpen, DosRead, DosChgFilePtr*
Family API: available
Errors to consider:

6 The handle must be one returned by *DosOpen* and not yet closed
33 The file is locked in the range where the write would fall
109 The handle specifies a pipe for which no read handles exist

(The following are seen only if the file was opened in the Return-errors mode.)

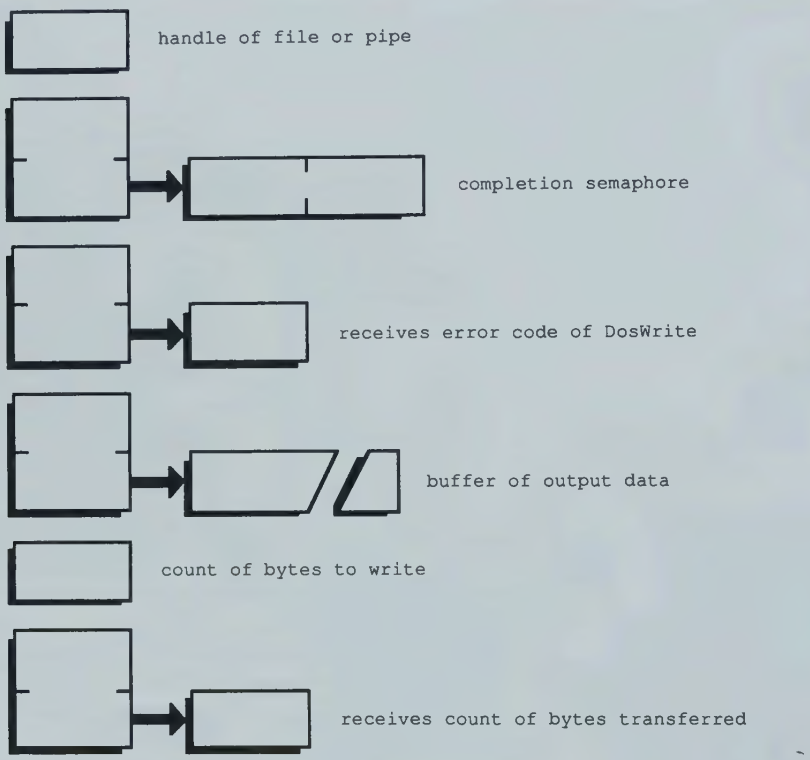
19 The disk is write-protected
21 Device not ready
22 Device driver does not do writes (e.g. MOUSE$)
23 Read-check error
25 Error on seek to track
27 Error on search to sector
29 Error on write

31 General failure reported by device driver
34 Disk volume has been changed since file was opened

```
extern unsigned far pascal DOSWRITE(
                        HANDLE,
                        char far *,
                        unsigned,
                        unsigned far *);
#define DosWrite DOSWRITE

function DosWrite       (     TheHandle : Handle;
                              TheData : ADSMEM;
                              SizeAsked : word;
                        vars SizeMoved : word
                        ) : word; external;
```

DosWriteAsync

Start asynchronous write to file



The system creates a thread to run asynchronously and perform a *DosWrite* operation. If the thread is created successfully, a code of zero is returned and the calling thread may process other things while the write is being done. The *DosWrite* operation writes the data and sets the count of bytes transferred. When it is finished, its return code is deposited in the third parameter. Then the semaphore is cleared and the created thread terminates. The proper sequence of operations is:

- *DosSemSet*
- *DosWriteAsync*
- Do work not involving the buffer space
- *DosSemWait*

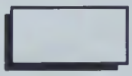
- Examine return codes and count

Discussion: *See* 8.11.5, 9.4.4 **Include:** FILES and ASYNC
Related: *DosWrite, DosOpen, DosSemSet, DosSemWait*
Family API: not available
Errors to consider:
155 unable to create a thread

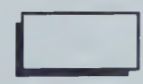
```
extern unsigned far pascal DOSWRITEASYNC(
                        HANDLE,
                        SEMHANDLE,
                        unsigned far *, /* retcode of DosRead */
                        char far *,     /* buffer */
                        unsigned,       /* bufsize */
                        unsigned far *  /* count read */
                        );
#define DosWriteAsync DOSWRITEASYNC

function DosWriteAsync (     TheHandle : Handle;
                             ASemaphore: SemHandle; { e.g. ADS MySem }
                        vars WriteRet  : word; { error code of DosWrite }
                             TheData   : ADSMEM; { e.g. ADS BufSpace[1] }
                             SizeAsked : word;
                        vars SizeMoved : word { size of buffer space }
                        ) : word; external;
```

DosWriteQueue

Put element into queue



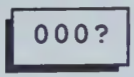
handle retrned by DosOpenQueue or DosCreateQueue



word describes item

word gives size of item

doubleword contains far pointer to item



priority of item or zero

If the queue specified by the first parameter has been closed by the process that

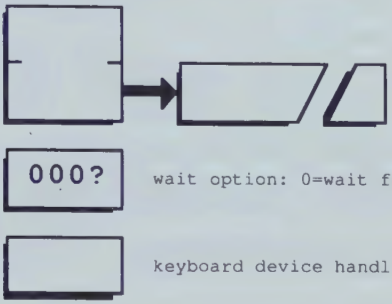
owned it, an error code is returned. Otherwise a four-word item is added to the queue. The conventional use of the words is as noted in the stack diagram: a descriptive word, a size word, and a far pointer. However, only the four words are written to the queue; no other data is copied and the fourth parameter is not used as an address by the system. A far pointer will not be meaningful in the context of another process's address space unless it addresses a segment shared through *DosGetSeg*. If the queue is not ordered by priority (option zero or 1 given to *DosCreateQueue*) then the last parameter is ignored. When the queue is priority-ordered, the last parameter specifies a priority value, where 15 puts the item at the head of the queue, zero puts it at the tail of the queue, and other priorities sort between (items are ordered FIFO within priority values). A priority value over 15 is set to 15.

Discussion: *See* 9.7.3, 9.7.4 **Include:** ASYNC
Related: *DosOpenQueue, DosCreateQueue*
Family API: not available
Errors to consider: none
337 The first parameter must specify an existing, open queue

```
extern unsigned far pascal DOSWRITEQUEUE(
                                HANDLE,
                                word, word,
                                void far *,
                                int);
#define DosWriteQueue DOSWRITEQUEUE

function DosWriteQueue( TheHdle : Handle;
                        TheDesc : word;
                        TheSize : word;
                        ThePtr  : ADSMEM; { use RETYPE(ADSMEM,x) }
                        ThePosn : integer
                        ):word; external;
```

KbdCharIn

Keyboard input



If the specified keyboard exists and has the keyboard focus, its input buffer is checked. If there is data ready, the first keystroke is removed and a data record describing it is written in the first parameter.

If no character is ready and the second parameter is zero, the current thread is suspended until a character is ready. Otherwise control returns at once with a zero return code and the keystroke record unchanged. (Fill the first parameter with zeros before the call; then a timestamp of zero may be taken as a sign of no data being available.)

If the keyboard is generating Double-byte characters, two calls are required to get the full character. Use the final-character flag in the NLS status bits (Figure 11.3, p. 335) to determine if a second byte is needed.

Discussion: *See* 11.6 **Include:** KEYBOARD
Related: *KbdPeek, KbdOpen, KbdGetFocus*
Family API: yes
Errors to consider:
375 The wait option must be 0 or 1
439 The keyboard handle must be 0 or one returned by *KbdOpen*
445 The keyboard focus is required
464 Detached processes have no keyboard access

```
extern unsigned far pascal KBDCHARIN(
                        struct KeyRecord far *,
                        int, /* KeyWAIT or KeyNOWAIT */
                        HANDLE);
#define KbdCharIn KBDCHARIN
```

```
function KbdCharIn      (vars TheKey: KeyRecord;
                             Option: integer; { Key_Wait or
Key_NoWait }
                             TheHdl: Handle
                        ) : word; external;
```

KbdClose

Release keyboard handle

handle received from KbdOpen

If the specified keyboard handle has the keyboard focus, the focus is released as for *KbdFreeFocus*. Any data in the keyboard's input buffer is flushed. The handle is released and may be reused by *KbdOpen*. Active keyboard handles are automatically closed when a process terminates.

Discussion: *See* 11.3, 11.3.1 **Include:** KEYBOARD
Related: *KbdOpen, KbdFreeFocus*
Family API: not available
Errors to consider:
439 The keyboard handle must be 0 or one returned by *KbdOpen*

```
extern unsigned far pascal KBDCLOSE(HANDLE);
#define KbdClose KBDCLOSE

function KbdClose       (TheHandle:Handle
                        ) : word; external;
```

KbdDeregister

Revoke replacement of keyboard subsystem

(no parameters)

If this process has registered a replacement keyboard subsystem with KbdRegister, it is disconnected and keyboard functions are once more handled by the OS/2 calls.

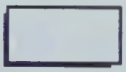
Discussion: *See* 11.9 **Include:** KEYBOARD

Related: *KbdRegister*
Family API: not available
Errors to consider:
411 No keyboard subsystem has been registered by this process

```
extern unsigned far pascal KBDDEREGISTER();
#define KbdDeregister KBDDEREGISTER

function KbdDeregister: word; external;
```

KbdFlushBuffer

Empty keyboard input buffer



keyboard device handle, zero for default

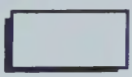
If the specified keyboard exists and has the keyboard focus, its input buffer is cleared of any unread input.

Discussion: *See* 11.3.3 **Include:** KEYBOARD
Related: *KbdCharIn, KbdStringIn*
Family API: available
Errors to consider:
439 The keyboard handle must be 0 or one returned by *KbdOpen*
445 The keyboard focus is required
464 Detached processes have no keyboard access

```
extern unsigned far pascal KBDFLUSHBUFFER(HANDLE);
#define KbdFlushBuffer KBDFLUSHBUFFER

function KbdFlushBuffer(TheHandle : Handle) : word; external;
```

KbdFreeFocus

Inactivate logical keyboard



handle received from KbdOpen, or zero for default

If the specified logical keyboard has the keyboard focus, it is given up. If other threads are waiting for the focus (having called *KbdGetFocus*), one of them

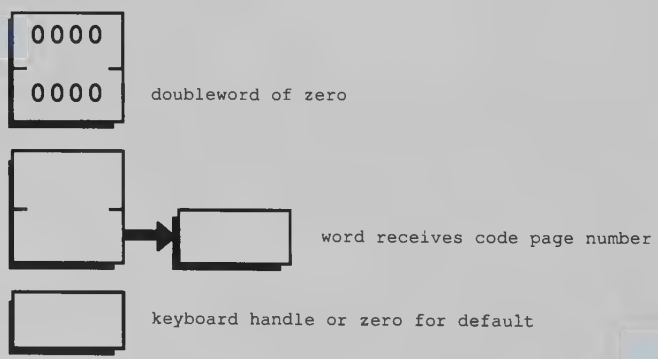
receives it. If none are waiting, keyboard input data is redirected to the default keyboard, but the focus remains available for any keyboard to take. This call is performed automatically if the keyboard handle is closed while holding the focus. Since the default keyboard may take the focus via *KbdGetFocus*, it may also release it with this call.

Discussion: *See* 11.3, 11.3.1 **Include**: KEYBOARD
Related: *KbdOpen, KbdClose, KbdGetFocus*
Family API: not available
Errors to consider:
439 The keyboard handle must be 0 or one returned by *KbdOpen*
445 The keyboard focus is required

```
extern unsigned far pascal KBDFREEFOCUS(HANDLE);
#define KbdFreeFocus KBDFREEFOCUS

function KbdFreeFocus  (TheHandle:Handle
                        ) : word; external;
```

KbdGetCP

Get number of keyboard code page



The number of the code page (translation table) being used with the specified logical keyboard is returned. Zero signifies the default code page built into the system. Other numbers are as set with the *codepage=* statement in the system configuration file, or the first word of a custom code page installed with *KbdSetCustXT*.

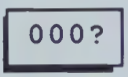
Discussion: *See* 11.8 **Include**: KEYBOARD
Related: *KbdSetCP, KbdSetCustXT*
Family API: not available

Errors to consider:
439 The keyboard handle must be 0 or one returned by *KbdOpen*
445 The keyboard focus is required

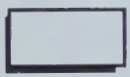
```
extern unsigned far pascal KBDGETCP(
                        long,           /* must be zero */
                        char far *,     /* code page image */
                        HANDLE);
#define KbdGetCP KBDGETCP

function KbdGetCP       (AnyZero : integer4;
                         CodePage: ADSMEM; { e.g. ADS MyCP }
                         TheHdl  : Handle
                         ) : word; external;
```

KbdGetFocus

Activate logical keyboard



wait option: 0=wait for focus, 1=do not wait



handle received from KbdOpen, or zero for default

The handle may be zero for the default logical keyboard, or a handle returned by *KbdOpen*. If no other logical keyboard (among all programs executing in this screen group) has the keyboard focus, it is given to the specified keyboard. Once the focus is on a keyboard, all input from the keyboard of this screen group will come to this logical keyboard. If some other keyboard has the focus, the wait option is tested. If its least significant bit is 1, control returns with an error code. If it is zero, the current thread is suspended in a queue of threads waiting for the focus. When the current holder releases it (with *KbdFreeFocus* or *KbdClose*) one of the waiting threads will receive the focus.

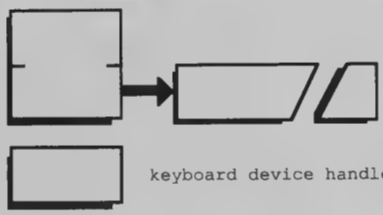
Discussion: *See* 11.3, 11.3.1 **Include:** KEYBOARD
Related: *KbdOpen, KbdClose, KbdFreeFocus*
Family API: not available
Errors to consider:
439 The keyboard handle must be 0 or one returned by *KbdOpen*
446 The focus is on another keyboard (and wait option was 1)
464 Detached programs have no keyboard access

```
extern unsigned far pascal KBDGETFOCUS(int HANDLE);
```

```
#define KbdGetFocus KBDGETFOCUS
                      (Option: integer;
function KbdGetFocus   (TheHandle : Handle
                       ) : word; external;
```

KbdGetStatus

Get status of logical keyboard



keyboard status structure, figure 11.1, p.331

keyboard device handle, zero for default

The Keyboard Status data structure passed (see Figure 11.1, p. 331) must have its length set in its first word. If the specified keyboard exists and has the keyboard focus, its status is returned in the structure. From the status information the program can learn the shift state, whether the mode is "raw" or "cooked," and what the "turnaround" (logical end of line in cooked mode, usually 0dh for Enter) character is. The data structure returned by this call is acceptable as input to *KbdSetStatus*.

Discussion: *See* 11.4 **Include:** KEYBOARD

Related: *KbdSetStatus, KbdOpen, KbdGetFocus*

Family API: available

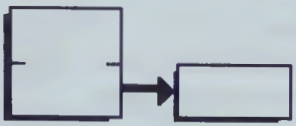
Errors to consider:

376 The structure length field must contain the structure length
439 The keyboard handle must be 0 or one returned by *KbdOpen*
445 The keyboard focus is required
464 Detached processes have no keyboard access

```
extern unsigned far pascal KBDGETSTATUS(struct KeyStatus far
*,HANDLE);
#define KbdGetStatus KBDGETSTATUS

function KbdGetStatus  (vars TheStuff : KeyStatus;
                             TheHandle: Handle
                       ) : word; external;
```

KbdOpen

Create handle for logical keyboard



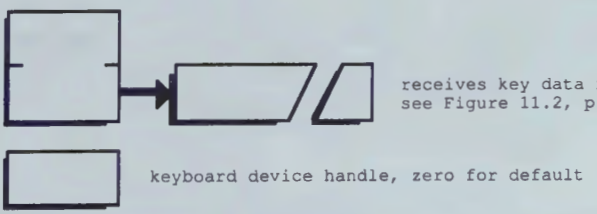
A handle is created and identified with a logical keyboard. The handle may be used only in the *Kbd* system calls, not in other calls that might take handles as parameters. The new keyboard is set to use the default code page set by the system configuration. The keyboard does not have the focus, and may not be used in any calls except *KbdClose* until the keyboard focus has been obtained with *KbdGetFocus*.

Discussion: *See* 11.3, 11.3.1 **Include**: KEYBOARD
Related: *KbdClose, KbdGetFocus*
Family API: not available
Errors to consider:
440 No more handles available
464 Detached programs have no keyboard access

```
extern unsigned far pascal KBDOPEN(HANDLE far *);
#define KbdOpen KBDOPEN

function KbdOpen        (vars NewHandle : Handle
                         ) : word; external;
```

KbdPeek

Sample keyboard input



If the specified keyboard exists and has the keyboard focus its input buffer is

sampled; if a keystroke is available, a data record describing it is copied into the first parameter. If the keyboard buffer is empty, no change is made in the first parameter. Clear it to zero before the call and use a zero timestamp as a sign of an empty buffer.

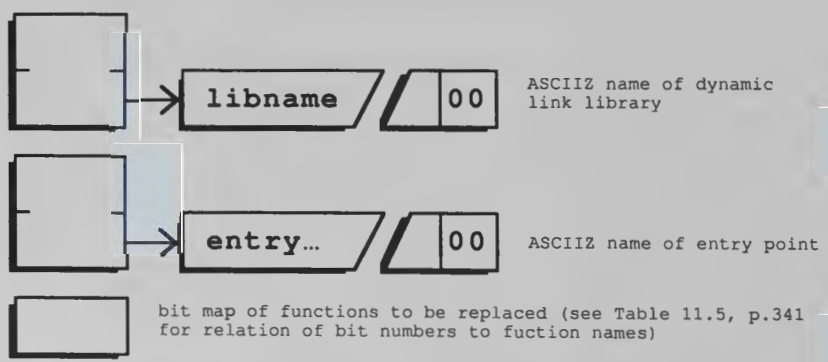
If the keyboard is generating Double-byte characters, two calls are required to sample the full character. Use the final-character flag in the NLS status bits (Figure 11.3, p. 335) to determine if a second byte is needed.

Discussion: *See* 11.6 **Include:** KEYBOARD
Related: *KbdCharIn, KbdOpen, KbdGetFocus*
Family API: available
Errors to consider:
439 The keyboard handle must be 0 or one returned by *KbdOpen*
445 The keyboard focus is required
464 Detached processes have no keyboard access

```
extern unsigned far pascal KBDPEEK(struct KeyRecord far *,HANDLE);
#define KbdPeek KBDPEEK

function KbdPeek        (vars TheKey : KeyRecord;
                             TheHdl : Handle
                         ) : word; external;
```

KbdRegister

Register replacement keyboard subsystem



The dynamic link library specified is loaded as for *DosLoadModule*, and the entry point named is looked up in it as for *DosGetProcAddr*. If this is cone successfully, the keyboard functions indicated by 1-bits in the bit map are

redirected so that, when one is called by any program in this screen group, control will pass instead to the entry point. The functions are numbered as shown in Table 11.5, p. 341, and the bits of the map are numbered in the conventional way with bit 0 the least significant of the word.

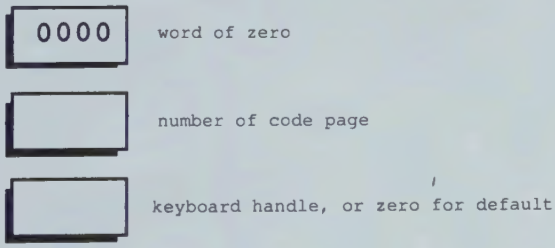
Only one keyboard subsystem may be registered in a screen group. *KbdDeregister* must be called to revoke the registration before another registration may be made.

Discussion: *See* 11.9 **Include:** KEYBOARD
Related: *KbdDeregister, KbdSynch*
Family API: not available
Errors to consider:
409 Bits 14 and 15 of the bit map must be zero
410 A registered subsystem is already active in this screen group

```
extern unsigned far pascal KBDREGISTER(
                        char far *,     /* libname */
                        char far *,     /* entry name */
                        OPTIONSET);
#define KbdRegister KBDREGISTER

function KbdRegister    (vars LibName, EntryName : char; { first of
string }
                        BitMap : word
                        ) : word; external;
```

KbdSetCP

Set predefined keyboard code page



If code page processing was initialized for the keyboard device during system configuration by a *devinfo kbd$...* statement, and if the specified code page was prepared with the *codepage=* statement, that code page is activated for the specified

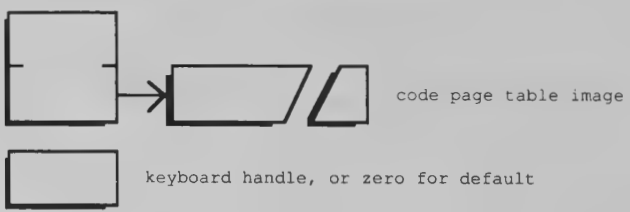
logical keyboard. When setting the keyboard translation it is usually a good idea to set a matching codepage number for the display hardware with *VioSetCP*.

Discussion: *See* 11.8 **Include:** KEYBOARD
Related: *KbdGetCP, KbdSetCustXT, KbdOpen, VioSetCP*
Family API: not available
Errors to consider:
439 The keyboard handle must be 0 or one returned by *KbdOpen*
442 The codepage couldn't be read from disk
443 The codepage number is invalid
444 Code page support was not set up during configuration
445 The keyboard focus is required
448 Selected code page has an invalid format

```
extern unsigned far pascal KBDSETCP(
                        int,            /* NOT long, must be zero */
                        int,            /* code page id */
                        HANDLE);
#define KbdSetCP KBDSETCP

function KbdSetCP       (AnyZero: integer;
                         CodePageNo: integer;
                         TheHdl : Handle
                         ) : word; external;
```

KbdSetCustXT

Set custom code page for logical keyboard



If code page support was set up with a *devinfo kbd$...* statement in the system configuration file, the format of the code page table is checked (not documented in this book). If it is satisfactory, the table is copied and put into effect on the specified keyboard.

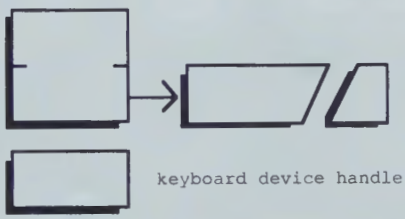
Discussion: *See* 11.8 **Include:** KEYBOARD
Related: *KbdGetCP, KbdSetCP, KbdOpen*

Family API: not available
Errors to consider:
439 The keyboard handle must be 0 or one returned by *KbdOpen*
448 Invalid code page format

```
extern unsigned far pascal KBDSETCUSTXT(
                        char far *, /* table image */
                        HANDLE);
#define KbdSetCustXT KBDSETCUSTXT

function KbdSetCustXT  (CodePage : ADSMEM; { e.g. ADS MyTable }
                        TheHdl : Handle
                        ) : word; external;
```

KbdSetStatus

Set status of logical keyboard



If the specified keyboard exists and has the keyboard focus, its status is set as required by the status mask in the Keyboard Status data structure (see Figure 11.2, p. 331). The best way to prepare the input data structure is to use *KbdGetStatus* and then change only the items of interest.

If neither Echo bit is set, Echo status isn't changed. If neither Mode bit is set, the Mode isn't changed. When changing the "turnaround" (logical end of line in cooked mode) note that the one or two bytes are treated as a 16-bit word, with the logical first (or only) byte in the lower address. If the shift state is changed, the keyboard LED indicators will reflect the new state.

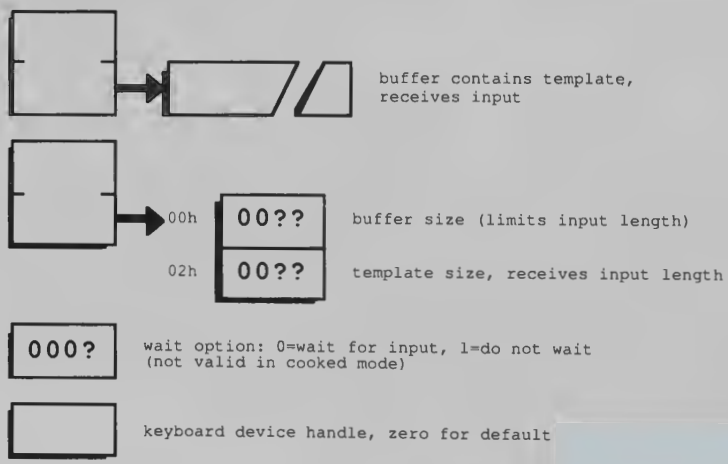
Discussion: *See* 11.4 **Include:** KEYBOARD
Related: *KbdGetStatus*
Family API: available
Errors to consider:
376 The shift state or another field of the structure is invalid
377 The Echo bits may not both be set

378 The Raw and Cooked bits may not both be set
439 The keyboard handle must be 0 or one returned by *KbdOpen*
445 The keyboard focus is required
464 Detached processes have no keyboard access

```
extern unsigned far pascal KBDSETSTATUS(struct KeyStatus far
*,HANDLE);
#define KbdSetStatus KBDSETSTATUS

function KbdSetStatus  (vars TheStuff: KeyStatus;
                            TheHandle: Handle
                        ) : word; external;
```

KbdStringIn

Read ASCII string



If the specified keyboard exists and has the keyboard focus, a string of extended ASCII characters is read from it into the buffer. The maximum buffer size is 255 bytes. The minimum is zero, which calls for no input at all. In all cases the count of bytes set in the buffer is returned in the second parameter.

Processing depends on the keyboard's mode, "raw" or "cooked." In raw mode the template size is ignored, but the wait option is tested. If wait is specified, control will not return until the buffer has been filled. In this case, it is possible that the last byte in the buffer might be the first of a two-byte extended-ASCII or Double-byte character. If no wait is specified, as many characters as are available

at the time, up to the limit, are copied into the buffer and control returns at once.

In cooked mode the wait option is ignored but the template length and template data are used. Available and incoming keystrokes are processed against the template until the Enter key (or other "turnaround" keystroke as set with *KbdSetStatus*) is received.

Discussion: *See* 11.4, 11.5 **Include:** KEYBOARD
Related: *KbdOpen, KbdGetFocus, KbdGetStatus, KbdSetStatus*
Family API: available
Errors to consider:
376 The buffer-size word may not exceed 255
439 The keyboard handle must be 0 or one returned by *KbdOpen*
445 The keyboard focus is required
464 Detached processes have no keyboard access

```
extern unsigned far pascal KBDSTRINGIN(
                        char far *,     /* buffer */
                        struct KeyBufLen far *,
                        int,            /* 0 wait/1 don't */
                        HANDLE);
#define KbdStringIn KBDSTRINGIN

function KbdStringIn    (vars TheBuffer  : char; {e.g. ADS MyBuf[1]}
                         vars TheSizes   : KeyBufLen;
                              WaitOpt    : integer; {0 wait/1 don't}
                              TheHandle  : Handle
                         ) : word; external;
```

KbdSynch

Serialize entry to replacement keyboard functions

000?	wait option: 0=wait, 1=return regardless

This operation is meant for use by a procedure which has been registered with *KbdRegister* as a replacement keyboard subsystem. It performs a *DosSemRequest* call on a semaphore that is unique to the screen group, thus serializing entry to the subsystem code.

Discussion: *See* 11.9 **Include:** KEYBOARD
Related: *KbdRegister*

Family API: not available
Errors to consider:
105 Previous owner of the semaphore has terminated
121 Semaphore not owned and wait option was 1 (zero wait timed out)

```
extern unsigned far pascal KBDSYNCH(int);
#define KbdSynch KBDSYNCH

function KbdSynch       (WaitOpt:integer
                         ) : word; external;
```

KbdXlate

Perform keystroke translation

translation data structure (see Figure 11.5, p.338)

keyboard handle, or zero for default

The data structure (Figure 11.5, p. 338) contains a shift state, a new scan code, and a translation state word. The valid input scan codes are shown in Table 11.3, p. 349. The state word should be set to zero to force a fresh start at translation; it will be reset to zero when a character is completed.

These input fields are processed against the code page of the specified keyboard and the results are set in the data structure. If the result is complete the translation flag word is set to 1. If more keystrokes are needed to resolve the character the flag word is set to zero and the translation state word is set nonzero.

A completed translation does not necessarily produce a character. The least significant six bits of the *KbdDDFlags* field must be consulted; if it yields a nonzero, see Table 11.4, p. 340 for the interpretation of the result. Only if the signal is zero will a translated ASCII character appear in the result.

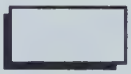
Discussion: *See* 11.8 **Include:** KEYBOARD
Related: *KbdGetCP, KbdSetCP, KbdSetCustXT*
Family API: not available
Errors to consider:
439 The keyboard handle must be 0 or one returned by *KbdOpen*

445 The keyboard focus is required
464 Detached processes have no keyboard access

```
extern unsigned far pascal KBDXLATE(
                        struct KeyXlate far *,
                        HANDLE);
#define KbdXlate KBDXLATE

function KbdXlate       (vars TheStuff:KeyXlate;
                             TheHdl: Handle
                        ) : word; external;
```

MouClose

Release mouse handle



handle returned by MouOpen

The handle must be one opened by *MouOpen*; it is released. If another handle is open to the mouse in this screen group, the mouse is unaffected. Otherwise if the mouse pointer is visible on the screen, it is removed, and the next time the mouse is opened it will be positioned in the center of the screen with a default pointer.

Discussion: *See* 12.5 **Include:** MOUSE
Related: *MouOpen*
Family API: not available
Errors to consider:
6 The handle is not an open one
22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUCLOSE(HANDLE);
#define MouClose MOUCLOSE

function MouClose (TheHdl : Handle) : word; external;
```

MouDeregister

Remove replacement mouse subsystem

(no parameters)

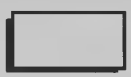
If this process has registered a replacement mouse subsystem with *MouRegister*, it is disconnected and mouse functions are again handled by the OS/2 calls.

Discussion: *See* 12.6 **Include**: MOUSE
Related: *MouRegister*
Family API: not available
Errors to consider:
416 This process hasn't registered a mouse subsystem

```
extern unsigned far pascal MOUDEREGISTER(void);
#define MouDeRegister MOUDEREGISTER

function MouDeRegister : word; external;
```

MouDrawPtr

Cancel removed region



handle returned by MouOpen

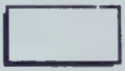
Any region of the screen established by *MouRemovePtr* (in which the mouse pointer may not be drawn) is cancelled. If the mouse pointer was invisible because of the region it is redrawn.

Discussion: *See* 12.3.5 **Include:** MOUSE
Related: *MouOpen, MouRemovePtr*
Family API: not available
Errors to consider:
6 The handle is not an open one
22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUDRAWPTR(HANDLE);
#define MouDrawPtr MOUDRAWPTR

function MouDrawPtr(TheHdl : Handle) : word; external;
```

MouFlushQue

Discard pending mouse events



handle returned by MouOpen

The event queue for the mouse for this screen group is emptied, discarding all pending events. The mouse position and mouse pointer are unchanged.

Discussion: *See* 12.4.2 **Include:** MOUSE

Related: *MouReadEventQue*

Family API: not available

Errors to consider:

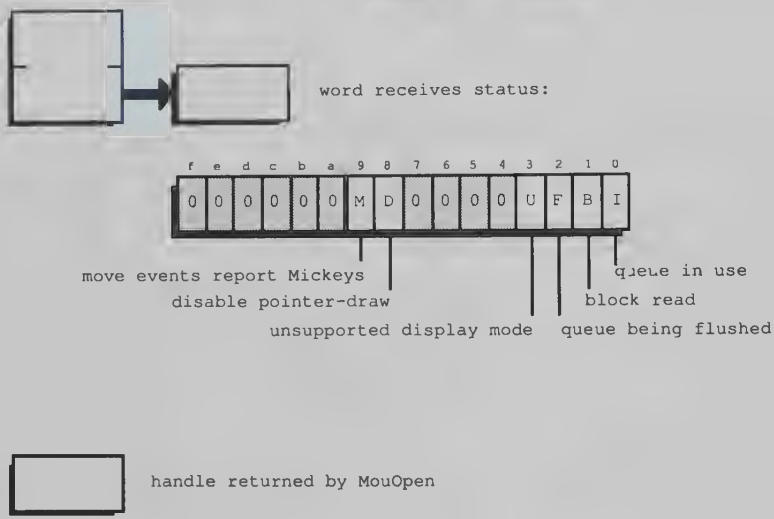
6 The handle is not an open one

22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUFLUSHQUE(HANDLE);
#define MouFlushQue MOUFLUSHQUE

function MouFlushQue(TheHdl:Handle) : word; external;
```

MouGetDevStatus

Get status flags for mouse



The state of the mouse for this screen group is returned. The word returned may be updated and used as input to *MouSetDevStatus*. Bits zero, one and two of the returned word reflect instantaneous conditions. Bit three is true when the pointer-drawing routine cannot cope with current display mode. Bits eight and nine are options set with *MouSetDevStatus* and are discussed there.

Discussion: *See* 12.3.2, 12.4.1.3 **Include:** MOUSE

Related: *MouOpen, MouSetDevStatus*

Family API: not available

Errors to consider:

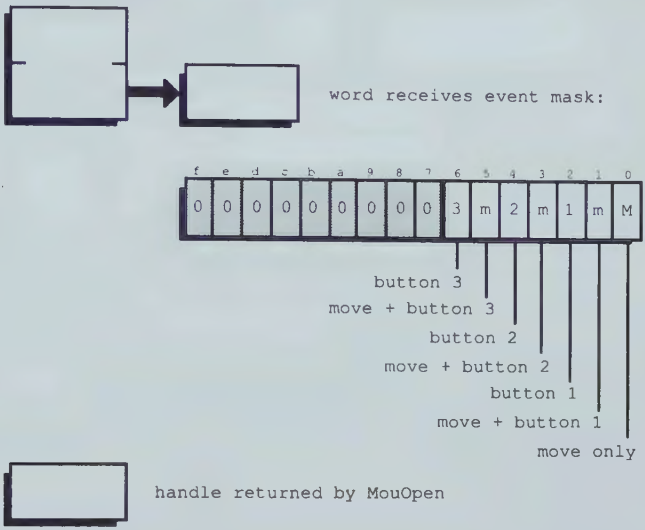
6 The handle is not an open one

22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUGETDEVSTATUS(unsigned far *,HANDLE);
#define MouGetDevStatus MOUGETDEVSTATUS

function MouGetDevStatus( vars TheStatus : word;
                               TheHdl : Handle
                          ) : word; external;
```

MouGetEventMask

Query possible mouse events



A set of bits, representing the events that can be posted to the mouse event queue for this screen group, is returned in the first parameter. The word may be updated and passed to *MouSetEventMask*.

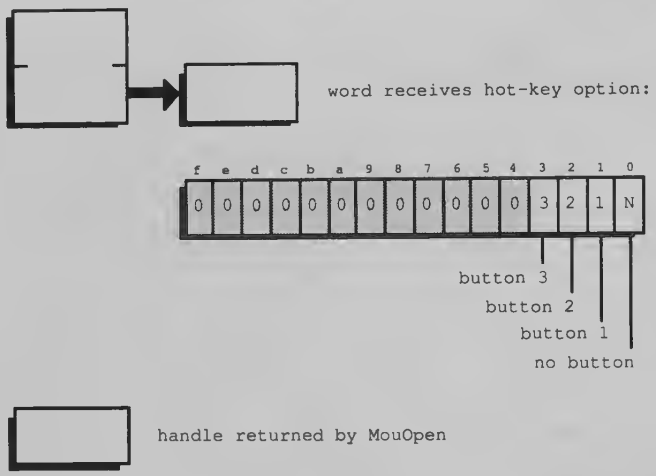
The initial event mask set by *MouOpen* contains bits for all possible events, but no impossible ones; that is, if the mouse has only two buttons, the "button 3" event bits will be zero.

Discussion: *See* 12.4.1.1 **Include:** MOUSE
Related: *MouOpen, MouSetEventMask*
Family API: not available
Errors to consider:
6 The handle is not an open one
22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUGETEVENTMASK(unsigned far *,HANDLE);
#define MouGetEventMask MOUGETEVENTMASK

function MouGetEventMask( vars TheMask : word;
                               TheHdl : Handle
                          ) : word; external;
```

MouGetHotKey

Query system hot-key signal



The session manager decides when it starts up if a mouse button or combination of buttons is to be a system hot-key, that is, call for a switch to the session manager's screen the way the Control-Escape keyboard signal does. Any process may query the decision with this call. If bit zero in the response is 1, there is no mouse-based hot-key signal. When bit zero contains zero, the combination of bits 1, 2 and 3 represents the mouse button combination that is a hot-key signal.

Only the process that initially sets the hot key combination may change it. Since that will always be the session manager, the *MouSetHotKey* function is not documented in this book.

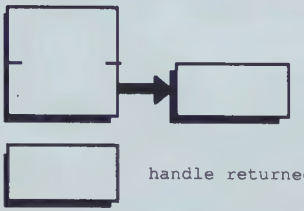
Discussion: *See* 12.3.1 **Include:** MOUSE
Related: *MouOpen*
Family API: not available
Errors to consider:
6 The handle is not an open one
22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUGETHOTKEY(unsigned far *,HANDLE);
#define MouGetHotKey MOUGETHOTKEY

function MouGetHotKey( vars TheKey : word;
```

```
                    TheHdl : Handle
                ) : word; external;
```

MouGetNumButtons

Query number of buttons on mouse



The number of buttons on the mouse is set in the first parameter. Ordinarily it will be one, three, or most commonly, two. The number of buttons is a hardware feature of the mouse; buttons can't be added or removed. Events concerning nonexistent buttons will not be listed in the default event mask returned by *MouGetEventMask*. Events concerning existing buttons, however, may be blocked with *MouSetEventMask*, effectively deadening the buttons.

Discussion: *See 12.3.1* **Include:** MOUSE

Related: *MouGetEventMask, MouOpen, MouSetEventMask*

Family API: not available

Errors to consider:

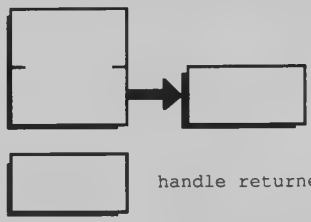
6 The handle is not an open one

22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUGETNUMBUTTONS(int far *,HANDLE);
#define MouGetNumButtons MOUGETNUMBUTTONS

function MouGetNumButtons( vars TheButtons : integer;
                                TheHdl : Handle
                           ) : word; external;
```

MouGetNumMickeys

Query units per centimeter



"Mickey" is the whimsical term for the smallest detectable increment of mouse movement; it is a hardware characteristic of the mouse. The number of Mickeys the mouse will report during a one-centimeter move is returned in the first parameter. When movements are reported in relative Mickeys (see *MouSetDevStatus*) this value can be used to measure absolute position of the mouse. The relation between Mickeys and screen units (character cells or pixels) is established by *MouSetScaleFact*.

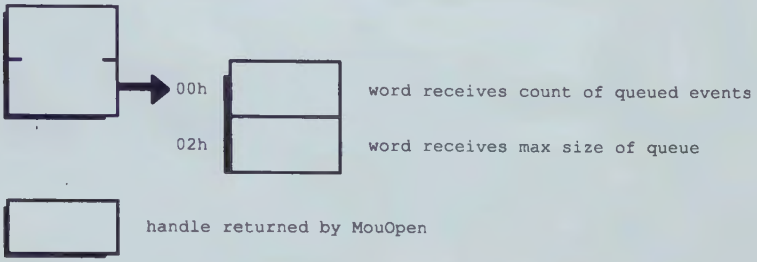
Discussion: *See* 12.3.1, 12.3.3 **Include**: MOUSE
Related: *MouGetScaleFact, MouOpen, MouSetDevStatus*
Family API: not available
Errors to consider:

6 The handle is not an open one
22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUGETNUMMICKEYS(int far *,HANDLE);
#define MouGetNumMickeys MOUGETNUMMICKEYS

function MouGetNumMickeys( vars TheMickeys : integer;
                                TheHdl : Handle
                           ) : word; external;
```

MouGetNumQueEl

Query state of event queue



The state of the mouse event queue is returned as two words in the first parameters. The first word receives the number of events presently in the queue; these events are available for reading with *MouReadEventQue* or the count can be cleared by *MouFlushQue*. The second word receives the size of the queue. This number is usually established by a parameter of the configuration file *device=* statement that loaded the mouse device driver. If the two numbers are equal, the queue is full and when a new event is added an old one will be lost.

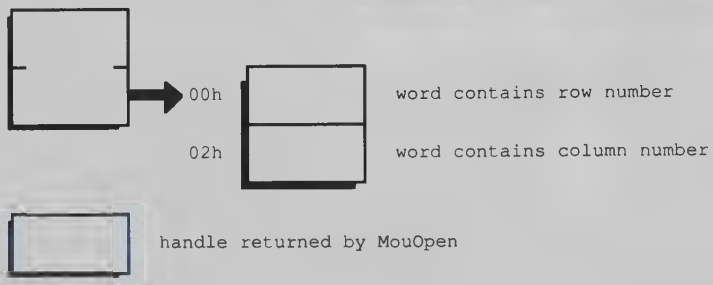
Discussion: *See* 12.4.2 **Include:** MOUSE
Related: *MouFlushQue, MouOpen, MouReadEventQue*
Family API: not available
Errors to consider:

6 The handle is not an open one
22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUGETNUMQUEEL(struct MQSTATUS far *,
HANDLE);
#define MouGetNumQueEl MOUGETNUMQUEEL

function MouGetNumQueEl( vars TheQStat : MQStatus;
                              TheHdl : Handle
                         ) : word; external;
```

MouGetPtrPos

Query mouse position



The screen position of the mouse is returned in the first parameter. The coordinates are in character rows and columns when the screen is in text mode, in pixel rows and columns when it is in a graphics mode. In either case, row zero column zero is the upper left corner of the screen. The mouse device driver tracks mouse position so long as the mouse is open, regardless of whether the mouse pointer is being drawn or what events are being queued. The relation between screen coordinates and Mickeys (increments of mouse movement) is set by *MouSetScaleFact*. A new screen position can be set with *MouSetPtrPos*.

Discussion: *See* 12.4.3 **Include:** MOUSE

Related: *MouGetScaleFact, MouOpen, MouSetPtrPos*

Family API: not available

Errors to consider:

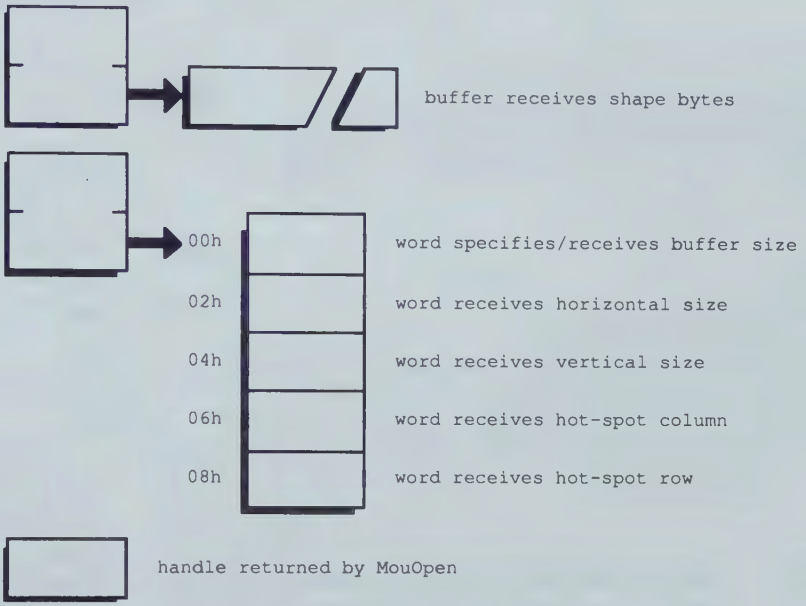
6 The handle is not an open one

22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUGETPTRPOS(struct MPTRPOS far *,
HANDLE);
#define MouGetPtrPos MOUGETPTRPOS

function MouGetPtrPos( vars ThePos : MPtrPos;
                           TheHdl : Handle
                      ) : word; external;
```

MouGetPtrShape

Query mouse pointer data



If the size specified in the first word of the second parameter is not sufficient to hold the bytes that control the appearance of the mouse pointer, the minimum size is set and an error code is returned. When the buffer is large enough, the shape bytes are copied into the first parameter and the pointer definition is stored in the fields of the second parameter. The form and use of this information is discussed in the text.

Discussion: *See* 12.3.4 **Include:** MOUSE

Related: *MouOpen, MouSetPtrShape*

Family API: not available

Errors to consider:

6 The handle is not an open one

22 The handle is not one opened by *MouOpen*

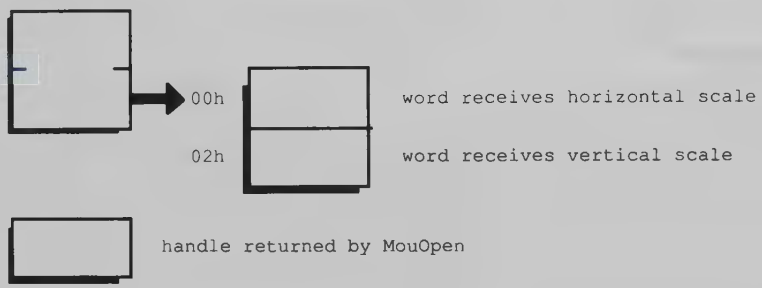
87 The buffer size in the second parameter is too small (the minimum buffer size has been set in that word)

```
extern unsigned far pascal MOUGETPTRSHAPE(char far *, /* buffer */
                                          struct MPTRDEF far *,
                                           HANDLE);
```

```
#define MouGetPtrShape MOUGETPTRSHAPE

function MouGetPtrShape(    ShapeBuff : ADSMEM; { ADS of whatever }
                         vars ThePDef  : MPtrDef;
                              TheHdl : Handle
                           ) : word; external;
```

MouGetScaleFact

Query ratio of mouse to pointer movement



The ratio between units of physical mouse motion (Mickeys) and visible motion of the mouse pointer in screen coordinates is returned in the first parameter. The ratio is initially set by *MouOpen* and may be changed by *MouSetScaleFact*. The first word receives the number of Mickeys that cause the mouse pointer to move one screen column (character cell or pixel, depending on screen mode). The second word receives the number of Mickeys that cause the mouse pointer to move one screen row.

Discussion: *See* 12.3.3 **Include:** MOUSE
Related: *MouOpen, MouGetNumMickeys, MouSetScaleFact*
Family API: not available
Errors to consider:
6 The handle is not an open one
22 The handle is not one opened by *MouOpen*

```
extern unsigned far pascal MOUGETSCALEFACT(struct MSCALE far
*,HANDLE);
#define MouGetScaleFact MOUGETSCALEFACT

function MouGetScaleFact( vars TheStuff : MScale;
                               TheHdl : Handle
                        ) : word; external;
```

MouOpen

Get handle for mouse

far pointer to driver name or doubleword of zero

word receives handle

If a mouse handle is already open for this process, this call creates a duplicate handle for the same mouse (see *DosDupHandle).* Otherwise it opens a new handle to the mouse if one exists. The first parameter specifies how the mouse pointer is to be drawn. If the parameter is doubleword of zero, the default pointer drawing routine will be used (this is the usual case). Otherwise the parameter must address an ASCIIZ string naming a device driver loaded with a *device=* statement in the configuration file; that routine will be called to remove or draw the pointer whenever the mouse moves. When the mouse is first opened in a screen group, it is initialized as follows. The event mask is set for all events possible for the actual mouse. The event queue is emptied. The mouse position is set to the center of the screen. The pointer shape is set to a default (in text mode, FF FF 00 77, resulting in a solid block of color). Default scale factors are set based on the screen mode. The whole screen is blocked as by *MouRemovePtr*; *MouDrawPtr* must be called to display the pointer the first time.

Discussion: *See* 12.2 **Include:** MOUSE
Related: *MouClose, MouDrawPtr, MouRemovePtr*
Family API: not available
Errors to consider:

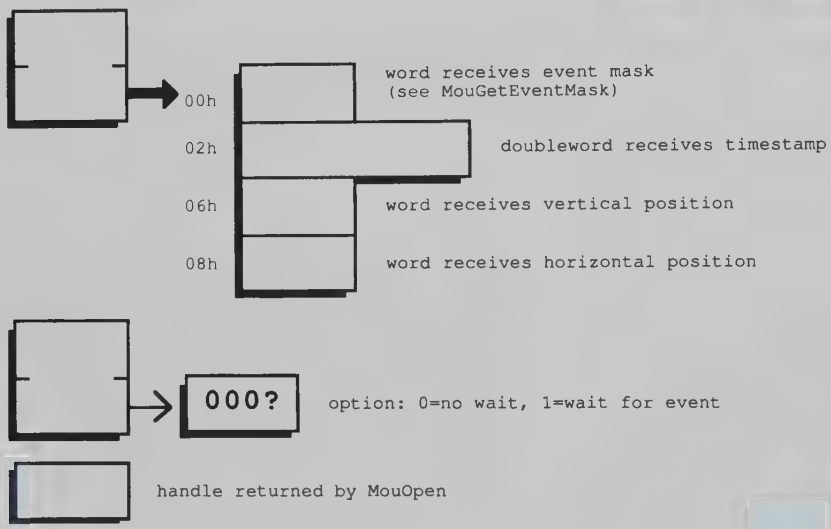
4 No handles are available, use *DosSetMaxFH* to create more
385 The mouse device driver was not started at configuration time
466 A detached process has no mouse available

```
extern unsigned far pascal MOUOPEN(char far *, HANDLE far *);
#define MouOpen MOUOPEN

function MouOpen (  DrawDriver : ADSMEM; { usually NilADS }
                    vars TheHdl : Handle
```

```
) : word; external;
```

MouReadEventQue

Get next mouse event



If the mouse event queue is empty, the second parameter is consulted; if it is zero, control returns at once with a return code of zero (prepare the event buffer with a default value to detect this case). Otherwise the calling thread is suspended until a mouse event occurs. The events that can be recognized are set by *MouSetEventMask*. When an event occurs, it is copied into the first parameter and control returns. The event mask field has the format returned by *MouGetEventMask*, with a bit set to indicate which event is reported. The timestamp is a copy of the system millisecond timer (see *DosGetInfoSeg*). The final two words of the event record may contain the absolute row and column position as from *MouGetPtrPos*, or may contain the relative number of Mickeys moved horizontally and vertically. This option is set by *MouSetDevStatus*.

Discussion: *See* 12.4.1 **Include:** MOUSE

Related: *MouOpen, MouSetDevStatus, MouSetEventMask*

Family API: not available

Errors to consider:

6 The handle is not an open one

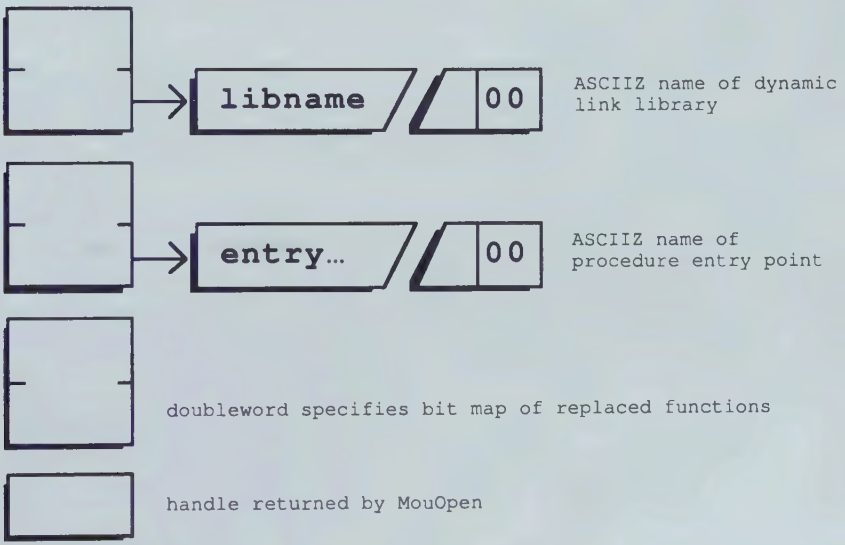
22 The handle is not one opened by *MouOpen*

435 The second parameter must be zero or 1

```
extern unsigned far pascal MOUREADEVENTQUE(struct MEVENT far *,
                                           int far *,
                                                   HANDLE);
#define MouReadEventQue MOUREADEVENTQUE

function MouReadEventQue(vars TheEvent : MEvent;
                         consts TheOption : integer;
                                TheHdl : Handle
                ) : word; external;
```

MouRegister

Register replacement mouse subsystem



The dynamic link library specified is loaded as for *DosLoadModule*, and the entry point named is looked up in it as for *DosGetProcAddr*. If this is done successfully, the display functions indicated by 1-bits in the bit map are redirected so that, when one is called by any process in this screen group, control will pass instead to the entry point. See the discussion topic for the requirements on the designated entry point. Table 12.2, p. 368 shows the relationship between replaced functions and bit numbers in the third parameter. Treat the third parameter as a 32-bit integer ("long" in C, "integer4" in Pascal) with bit 0 the

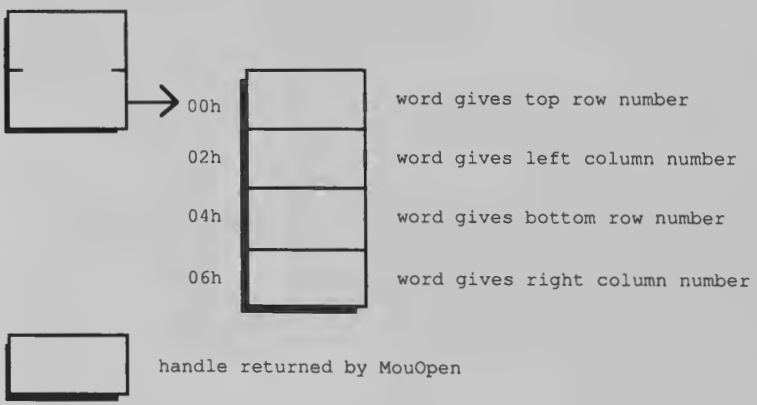
least significant, bit 31 the most significant; in assembly language push bits 31-16 first, bits 15-0 second. Only one mouse subsystem may be registered in a screen group. *MouDeregister* must be called to revoke the registration before another registration may be made.

Discussion: *See* 12.6 **Include:** MOUSE
Related: *MouDeregister, MouOpen, MouSynch*
Family API: not available
Errors to consider:
2 The dynamic link library couldn't be found
127 The entry point can't be found in the library module
414 Bits 22 through 31 of the doubleword mask must be zero, and the mask must be nonzero
415 Only one subsystem may be registered per screen group

```
extern unsigned far pascal MOUREGISTER(char far *, /* modname */
                                       char far *, /* entry name */
                                       unsigned long); /* bitmask */

#define MouRegister MOUREGISTER

function MouRegister( vars TheModule : char; { e.g. ModName[1] }
                      vars TheEntry  : char; { e.g. EntName[1] }
                           BitMask   : integer4
                     ) : word; external;
```

MouRemovePtr

Block mouse pointer display



The first parameter specifies a rectangular region of the screen in which the mouse pointer may not be drawn. The coordinates are in character cells when the screen is in text mode, in pixels in graphics modes. The region may have any size up the dimensions of the screen. (The screen dimensions are available from *VioGetMode*.) Whenever the mouse position is within the removed region the pointer will not be drawn. Outside the region the pointer will be drawn if the device status permits it (see *MouSetDevStatus*). Even within the region the mouse pointer position is updated and mouse events are recognized. Only one removed region may be defined in a screen group at one time. The *MouOpen* function sets the entire screen as a removed region. The region specified in this call will replace any prior region. The *MouDrawPtr* function cancels any removed region.

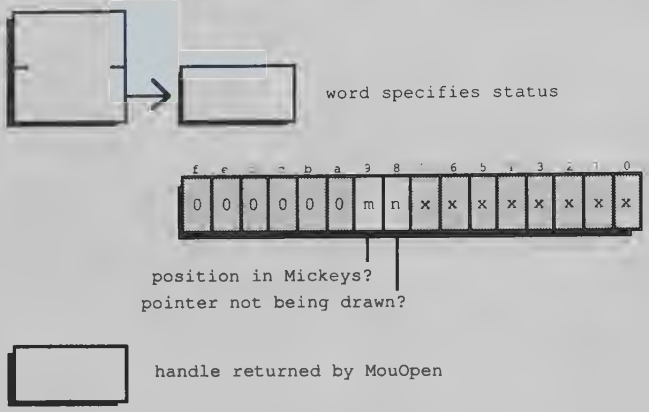
Discussion: *See* 12.3.5 **Include:** MOUSE
Related: *MouDrawPtr, MouOpen, MouSetDevStatus*
Family API: not available
Errors to consider:

6	The handle is not an open one
22	The handle is not one opened by *MouOpen*
87	The region coordinates may not exceed the size of the screen

```
extern unsigned far pascal MOUREMOVEPTR(struct MCOLLISION far
*,HANDLE);
#define MouRemovePtr MOUREMOVEPTR

function MouRemovePtr( consts TheArea : MCollision;
                                TheHdl : Handle
                      ) : word; external;
```

MouSetDevStatus

Set mouse options



Bits eight and nine of the first parameter are set as processing options. The word may be the one returned by *MouGetDevStatus*. Bits seven through zero are ignored; other bits must contain zero. Bit eight controls the drawing of the mouse pointer. When it is true the mouse pointer is removed from the display if it was visible and thereafter is not drawn each time the mouse is moved. When the bit is set false, the pointer drawing routine is called to redraw the pointer following each ensuing mouse move. This bit can be used to switch the pointer off temporarily (for instance while scrolling the screen) or permanently if the application wants to maintain its own pointer display. Bit nine controls the encoding of mouse movements in the event record returned by *MouReadEventQue*. When the bit is true, an event record contains the relative number of Mickeys moved vertically and horizontally. When the bit is false, an event record contains the absolute mouse position in screen coordinates.

Discussion: *See* 12.3.2, 12.4.1.3 **Include:** MOUSE
Related: *MouGetDevStatus, MouOpen, MouReadEventQue, MouGetPtrPos*
Family API: not available
Errors to consider:

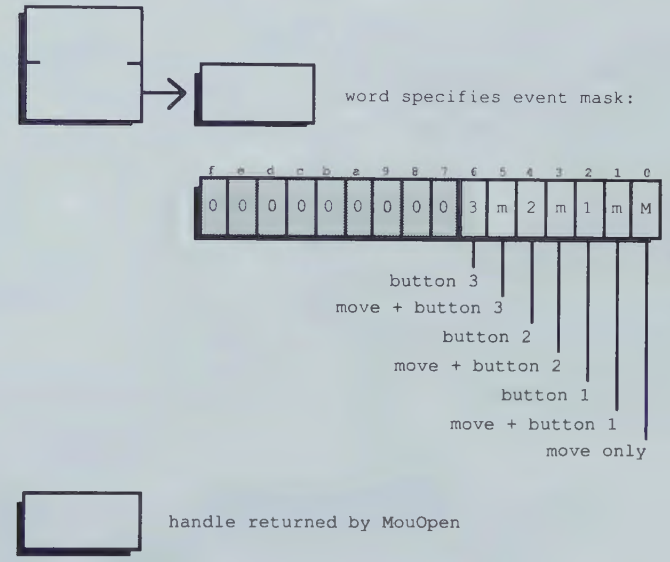
6	The handle is not an open one
22	The handle is not one opened by *MouOpen*
87	Bits 15 through 10 must be zero

```
extern unsigned far pascal MOUSETDEVSTATUS(unsigned far *,HANDLE);
#define MouSetDevStatus MOUSETDEVSTATUS
```

```
function MouSetDevStatus(consts TheStatus : word;
                                TheHdl : Handle
                        ) : word; external;
```

MouSetEventMask

Set possible mouse events



Bits six through zero of the first parameter establish what kinds of happenings will constitute a mouse "event" to be placed in the event queue and be read by *MouReadEventQue*. Bits 15 through 7 must contain zero. The current event mask is returned by *MouGetEventMask*, and the recommended way of setting a new mask is to retrieve the old one, modify only the bits of interest, and use it in this call. The mask given here selects which events will be queued for reading, but all moves are still noted by the device driver. The mouse position is kept up to date and the mouse pointer continues to be visible where it has not been removed.

Discussion: *See* 12.4.1.1 **Include**: MOUSE

Related: *MouGetEventMask, MouOpen, MouReadEventQue*

Family API: not available

Errors to consider:

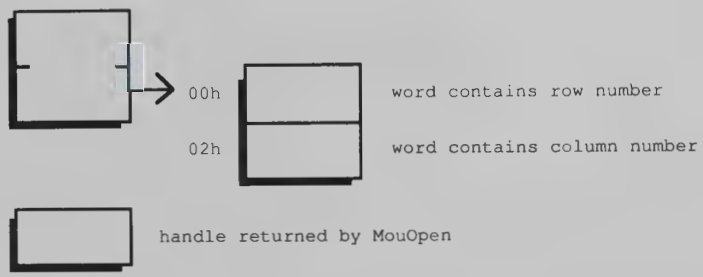
6 The handle is not an open one

22 The handle is not one opened by *MouOpen*
87 Bits 15 through 7 must be zero

```
extern unsigned far pascal MOUSETEVENTMASK(unsigned far *,HANDLE);
#define MouSetEventMask MOUSETEVENTMASK

function MouSetEventMask( consts TheMask : word;
                                TheHdl : Handle
                        ) : word; external;
```

MouSetPtrPos

Set mouse position



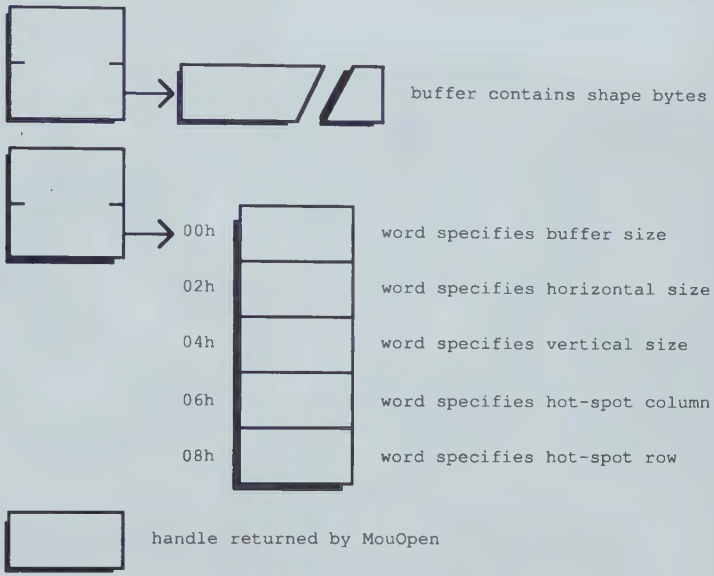
If the specified coordinates fall within the dimensions of the screen in its current mode they are set as the current mouse position. The upper left corner of the screen is always row zero, column zero. The screen dimensions are available from *VioGetMode*. If the mouse pointer is visible it is erased and redrawn.

Discussion: *See* 12.4.3 **Include:** MOUSE
Related: *MouGetPtrPos, MouGetScaleFact, MouOpen*
Family API: not available
Errors to consider:
6 The handle is not an open one
22 The handle is not one opened by *MouOpen*
87 The coordinates must fall in the screen area

```
extern unsigned far pascal MOUSETPTRPOS(struct MPTRPOS far *,HANDLE);
#define MouSetPtrPos MOUSETPTRPOS

function MouSetPtrPos(consts ThePos : MPtrPos;
                             TheHdl : Handle
                      ) : word; external;
```

MouSetPtrShape

Set mouse pointer appearance



The form and content of the pointer shape data is discussed in the text. When the screen is in text mode the buffer size must be 4 and the horizontal and vertical sizes must both be 1. In graphics modes the horizontal and vertical sizes must not exceed the screen size and the buffer size must be consistent with them. In text mode the hot-spot column and row are ignored; in graphics modes these numbers must be less than or equal to the horizontal and vertical sizes respectively. If the values are acceptable, they are passed to the pointer drawing code and will be used when the mouse pointer is next redrawn.

Discussion: *See* 12.3.4 **Include:** MOUSE

Related: *MouGetPtrShape, MouOpen*

Family API: not available

Errors to consider:

6 The handle is not an open one

22 The handle is not one opened by *MouOpen*

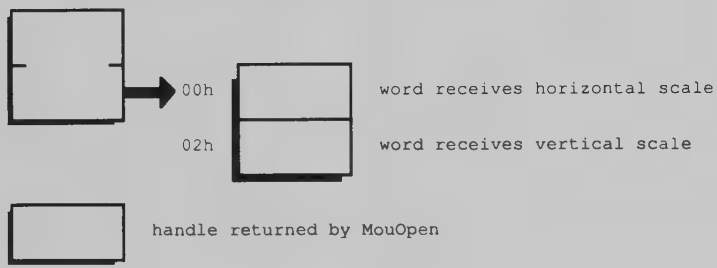
87 At least one of the fields of the second parameter is wrong

```
extern unsigned far pascal MOUSETPTRSHAPE(char far *, /* buffer */
                                          struct MPTRDEF far *,
                                          HANDLE);
```

```
#define MouSetPtrShape MOUSETPTRSHAPE

function MouSetPtrShape(     ShapeBuff : ADSMEM; { ADS of whatever }
                         consts ThePDef : MPtrDef;
                                TheHdl : Handle
                ) : word; external;
```

MouSetScaleFact

Set ratio of mouse to pointer movement



The ratio between units of physical mouse motion (Mickeys) and visible motion of the mouse pointer in screen coordinates is set from the first parameter. The ratio is initially set by *MouOpen* and may be queried with *MouGetScaleFact*. The first word specifies the number of Mickeys that cause the mouse pointer to move one screen column (character cell or pixel, depending on screen mode). The second word specifies the number of Mickeys that cause the mouse pointer to move one screen row.

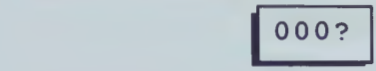
Discussion: *See* 12.3.3 **Include:** MOUSE
Related: *MouGetScaleFact, MouOpen*
Family API: not available
Errors to consider:

6	The handle is not an open one
22	The handle is not one opened by *MouOpen*
87	Both values must be nonzero

```
extern unsigned far pascal MOUSETSCALEFACT(struct MSCALE far
*,HANDLE);
#define MouSetScaleFact MOUSETSCALEFACT

function MouSetScaleFact( consts TheStuff : MScale;
                                  TheHdl : Handle
                        ) : word; external;
```

MouSynch

Serialize entry to replacement mouse subsystem



This operation is meant for use by a procedure which has been registered with *MouRegister* as a replacement mouse subsystem. It performs a *DosSemRequest* call on a semaphore that is unique to the screen group, thus serializing entry to the subsystem code. Note that the sense of the wait option is reversed from the similar *KbdSynch* function.

Discussion: *See* 12.6 **Include:** MOUSE
Related: *MouRegister*
Family API: not available
Errors to consider:
(Errors of *DosSemRequest*)

```
extern unsigned far pascal MOUSYNCH(int);
#define MouSynch MOUSYNCH

function MouSynch(WaitOpt:integer) : word; external;
```

VioDeregister

Remove a replacement video subsystem

(no parameters)

If this process had registered a set of replacement display functions with *VioRegister*, the replacement is cancelled and display functions are once more handled by the standard OS/2 calls.

Discussion: *See* 10.6 **Include:** VIOEO
Related: *VioRegister*
Family API: not available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
404 No display subsystem was registered by this process

```
extern unsigned far pascal VIODEREGISTER();
#define VioDeregister VIODEREGISTER

function VioDeregister : word; external;
```

VioEndPopup

End pop-up display

(no parameters)

If this process had begun a pop-up display with *VioPopup*, the pop-up is terminated. The pop-up screen group is released for the use of any other process that might be waiting for it. The physical display buffer contents are restored to their state before the pop-up. If the pop-up had preempted a graphics application, its *VioModeWait* watchdog thread is activated to restore the hardware display mode.

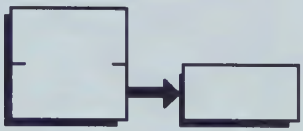
Discussion: *See* 10.4.4 **Include:** VIOEO
Related: *VioPopup; VioModeWait*
Family API: not available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
405 No pop-up active

```
extern unsigned far pascal VIOENDPOPUP();
#define VioEndPopup VIOENDPOPUP
```

```
function VioEndPopup : word; external;
```

VioGetANSI

Query ANSI display support



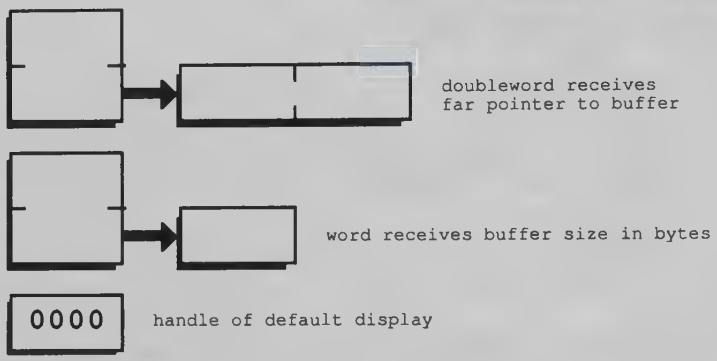
If support for ANSI escape sequences (in the display output stream written via *VioWrtTTY*) is active, a 1 is returned; otherwise a zero is returned. ANSI support can be inactivated or reactivated using *VioSetANSI*.

Discussion: *See* 10.3.4 **Include:** VIDEO
Related: *VioSetANSI*
Family API: not available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOGETANSI(
                        int far *,
                        HANDLE);        /* must be zero */
#define VioGetANSI VIOGETANSI

function VioGetANSI     (vars ANSIFlag : integer;
                         ZeroHdl : Handle
                         ) : word; external;
```

VioGetBuf

Get access to logical display buffer



The segment that contains the logical display buffer for this screen group is made part of the address space of this process, and a far pointer to the buffer is returned in the first parameter. The buffer's size in bytes is returned in the second parameter; this will be the number of screen rows multiplied by screen columns, times two.

One logical display buffer is shared by all processes in the screen group. All *VioWrt...* calls deposit their output in it. It is copied into the physical display buffer whenever this screen group is in the foreground and the system knows it has been updated. Use *VioShowBuf* to tell the system when data has been placed directly in the buffer.

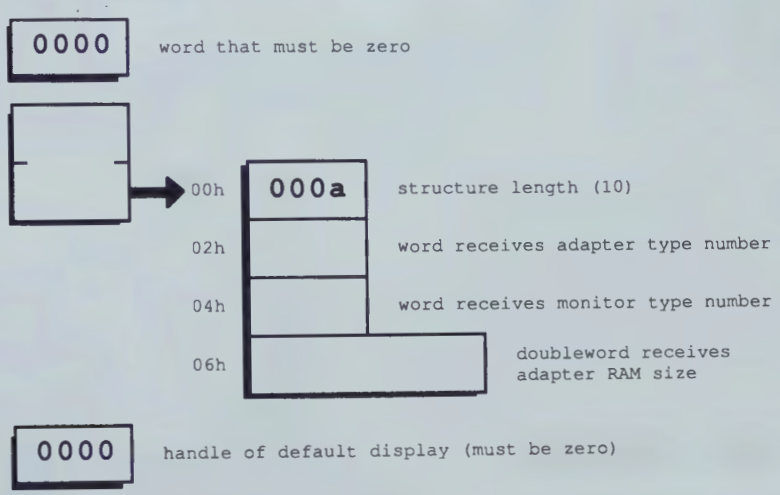
Discussion: *See* 10.4.3 **Include:** VIDEO
Related: *VioShowBuf, VioGetPhysBuf*
Family API: available
During Pop-up: not availabl **Presentation Manager:** available
Errors to consider:
355 Display is in a graphics mode
430 This may not be done during a pop-up display
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOGETBUF(
                        ADSMEM far *,
                        unsigned far *,
                        HANDLE);        /* must be zero */
#define VioGetBuf VIOGETBUF
```

```
function VioGetBuf      (vars BufADS: ADSMEM;
                         vars BufLen: word;
                         ZeroHdl : Handle
                         ) : word; external;
```

VioGetConfig

Query display hardware configuration



Information on the display hardware, as discovered while the system was booting up, is returned. The first word of the structure must be 4, 6 or 10. Only as many fields are returned as will fit. The information depends in part on switch settings and on hardware configuration data that can be wrong. The adapter type may be:

- 0 for an MDA
- 1 for a CGA
- 2 for an EGA
- 3 for a VGA
- 4 for an IBM 8514/A

The monitor type number may be:

- 0 for a monochrome monitor
- 1 for an RGB color display
- 2 for an enhanced color display
- 3 for an IBM 8503
- 4 for a VGA display
- 9 for an IBM 8514

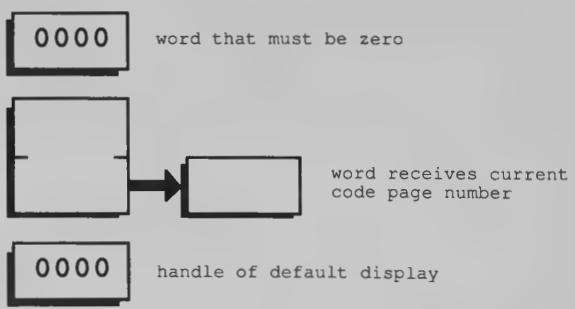
The amount of RAM installed in an EGA or VGA adapter is returned as a 32-bit integer.

Discussion: *See* 10.4.1.1 **Include:** VIDEO
Related: *VioGetMode*
Family API: available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
421 The first parameter must be zero
436 The display handle must be zero
438 The structure length field must be 4, 6 or 10
465 A detached process has no display access

```
extern unsigned far pascal VIOGETCONFIG(
                        int, /* must be 0 */
                        struct VioConfig far *,
                        HANDLE);        /* must be zero */
#define VioGetConfig VIOGETCONFIG

function VioGetConfig  (    AnyZero : integer; { must be zero }
                        vars Results: VioConfig;
                             ZeroHdl : Handle
                        ) : word; external;
```

VioGetCP

Query display codepage



A number is returned to identify the codepage (symbol set) currently in use for the display of this screen group. The number will be zero if the default codepage

is active. Otherwise it will be one of the numbers specified in the *codepage=* statement in the system configuration file or a number set by *VioSetCP*.

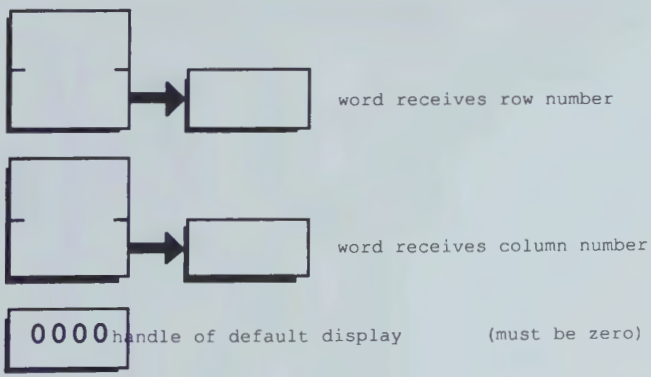
If a program-defined font has been installed with *VioSetFont*, a nonzero error code is returned. A program-defined font can be cleared with *VioSetCP*.

Discussion: *See* 10.4.1.5 **Include:** VIDEO
Related: *VioSetCP*, *VioGetFont*
Family API: available
During Pop-up: not available **Presentation Manager:** available
Errors to consider:
436 The display handle must be zero
465 A detached process has no display access
468 Program-defined font is active

```
extern unsigned far pascal VIOGETCP(
                        int, /* must be 0 */
                        int far *,
                        HANDLE);        /* must be zero */
#define VioGetCP VIOGETCP

function VioGetCP       (AnyZero : integer; { must be zero }
                        vars CPId : integer;
                        ZeroHdl : Handle
                        ) : word; external;
```

VioGetCurPos

Query cursor position in display buffer



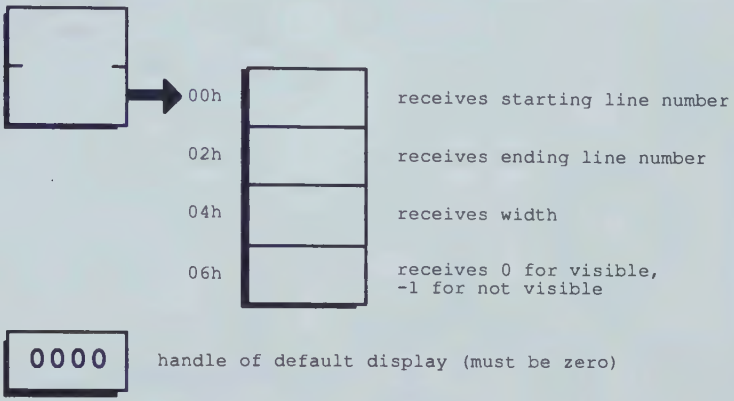
The position of the cursor in the logical display buffer for this screen group is returned as a row and a column number. The top left corner of the screen is at row zero and column zero. This is a logical cursor position maintained by the display output routines. It only reflects the actual display cursor when the current screen group is visible, the display is in a text mode, and no process is manipulating the display hardware directly.

Discussion: *See* 10.4.1.3 **Include:** VIDEO
Related: *VioSetCurPos, VioGetMode*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOGETCURPOS(
                        int far *, /* row */
                        int far *, /* col */
                        HANDLE);        /* must be zero */
#define VioGetCurPos VIOGETCURPOS

function VioGetCurPos  (vars TheRow : integer;
                        vars TheCol : integer;
                            ZeroHdl : Handle
                        ) : word; external;
```

VioGetCurType

Query appearance of cursor



The appearance of the text-mode cursor for this screen group is returned in a series of four words. The starting and ending line numbers describe the shape of the cursor. The units are lines of pels within a single character box, with line zero at the top of the box; hence the usual underscore cursor in a font with 8x14 characters starts on line 12 and ends on line 13. If the starting line exceeds the ending line the cursor "wraps around" the box from bottom to top. (The number of rows in a character box in the active font can be queried with *VioGetFont*.) The fourth field receives zero if the cursor is visible on the screen and -1 (0ffffh) if it is not. The cursor width field is returned as zero in text modes.

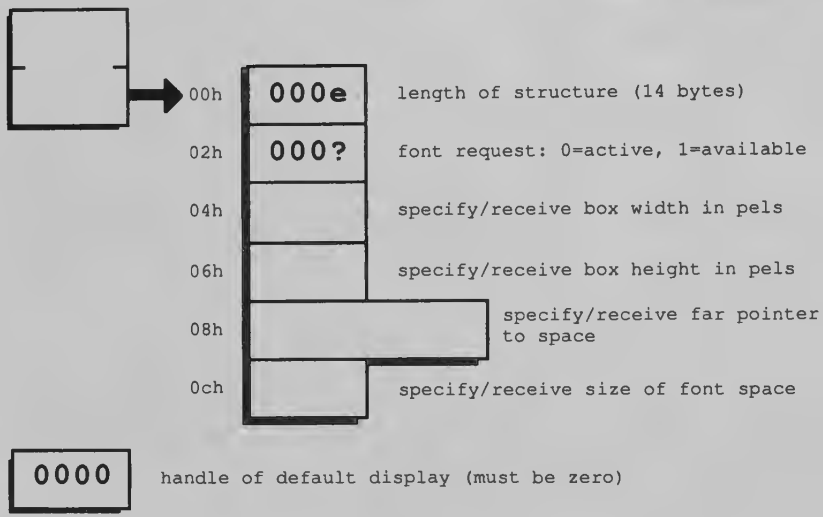
Discussion: *See* 10.4.1.4 **Include**: VIDEO
Related: *VioSetCurType*, *VioGetCurPos*, *VioGetMode*, *VioGetFont*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOGETCURTYPE(
                        struct VioCurType far *,
                        HANDLE);        /* must be zero */
#define VioGetCurType VIOGETCURTYPE

function VioGetCurType (vars TheStuff: VioCurType;
                        ZeroHdl : Handle
```

```
) : word; external;
```

VioGetFont

Examine the display symbol font



Information about a display font is returned. If the far pointer in the fifth field is specified as zero, a far pointer of the font image will be supplied by the system. This is usually the address of a segment of storage allocated for the purpose of holding a copy of a font, but in some cases it will be the address of a font image in ROM. If a nonzero pointer is specified in the fifth field, and a length at least as great as the size of the font is specified in the last field, the font will be copied into the addressed storage.

To examine the active font, specify zero in the second word. The width and height, in pels, of a character box are returned in the third and fourth fields. This type of request is only supported when the display adapter is an EGA or VGA. (The adapter type can be determined with *VioConfig*.) To examine a specific font from the available set, specify 1 in the second field and specify the box dimensions of the desired font in the third and fourth fields (8x8, 8x14, 9x14 or 9x16). The set of available fonts may include fonts in ROM, default fonts built into the device driver, and fonts loaded from a file specified in a *devinfo* statement in the configuration file. Some fonts are derived by expanding others; for instance a 9x16 font might be derived from a 9x14 font.

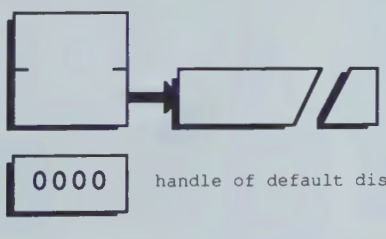
A font image contains 256 bit maps, one to define a symbol for every possible byte value. Each bit map contains one byte for each row in a character box. Where a byte contains a 1-bit the corresponding pel will be shown in the foreground color; a zero-bit pel will display in background color. A modified font can be activated with *VioSetFont* using this same data structure as input. Since a font could be in ROM, it should be copied before modification.

Discussion: *See* 10.4.1.5 **Include:** VIDEO
Related: *VioSetFont, VioGetMode, VioGetCP, VioSetCP*
Family API: available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
436 The display handle must be zero
438 Structure length must be 14 and the font space must be large enough for the entire font (256 times number of rows in a box).
465 A detached process has no display access
467 No font available of the requested dimensions

```
extern unsigned far pascal VIOGETFONT(
                        struct VioFont far *,
                        HANDLE);        /* must be zero */
#define VioGetFont VIOGETFONT

function VioGetFont     (vars TheStuff : VioFont;
                             ZeroHdl : Handle
                        ) : word; external;
```

VioGetMode

Query display mode



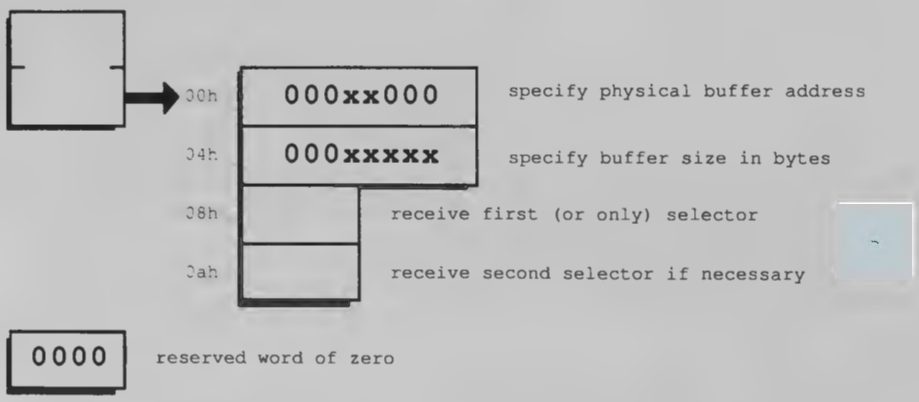
Information on the display mode for this screen group is returned. The fields of the data structure are discussed in detail in the text. The same structure can be used as input to *VioSetMode*.

Discussion: *See* 10.4.1.2 **Include:** VIDEO
Related: *VioSetMode*, *VioGetConfig*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOGETMODE(
                        struct VioMode far *,
                        HANDLE);        /* must be zero */
#define VioGetMode VIOGETMODE

function VioGetMode     (vars TheStuff : VioMode;
                             ZeroHdl : Handle
                        ) : word; external;
```

VioGetPhysBuf

Get address of physical display buffer



All or part of the physical display buffer is made part of the address space of this process. The starting buffer address is specified as a 32-bit integer, for example 0b8000h or 0a0000h, not in selector:offset form. The length of buffer space is specified as a 32-bit integer; if it exceeds 0ffffh two selectors will be returned.

The physical address must be greater than or equal to 0a0000h. The sum of address and length must be less than 0c0000h. The process has write access to the display buffer at all times, but ought not to update the buffer unless it has

been locked with *VioScrLock*. Otherwise there is a possibility of overwriting the display of some other screen group.

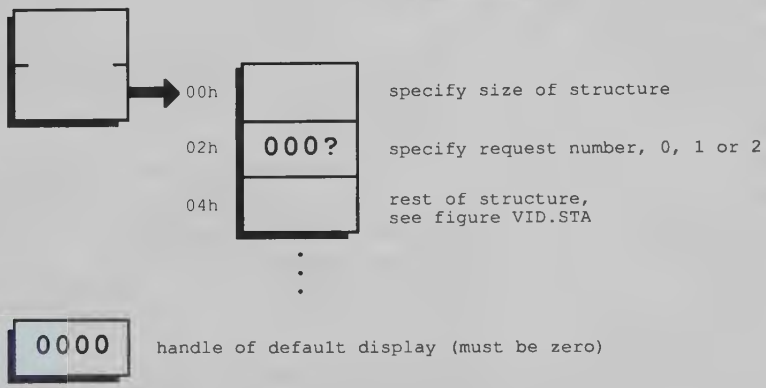
Access to the physical buffer is only required to support graphics display modes. In text display modes it is almost as fast and much more convenient to operate in the logical display buffer returned by *VioGetBuf*.

Discussion: *See* 10.5.2 **Include:** VIDEO
Related: *VioScrLock, VioScrUnlock, VioGetBuf*
Family API: available:
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
350 The address or address plus length falls outside physical buffer
429 This may only be done when the screen group is in the foreground
430 This may not be done during a pop-up display
436 The second parameter must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOGETPHYSBUF(
                        struct VioPhysBuf far *,
                        HANDLE);        /* must be zero */
#define VioGetPhysBuf VIOGETPHYSBUF

function VioGetPhysBuf (vars TheStuff : VioPhysBuf;
                             ZeroHdl : Handle
                        ) : word; external;
```

VioGetState

Query display hardware settings



One of three hardware settings is returned depending on the number in the second field. In all cases, the size in the first word must be exact for the type of request and amount of data to be returned. When the request number is 0, the palette register contents are returned. The third word specifies the number of the first palette register to be returned, from 0 to 15. The size in the first word implies how many registers are to be returned. (Most commonly the size is 38, the first register is 0, and all 16 registers are returned.)

When the request number is 1 and the size is 6, the color number of border area of the text-mode display is returned in the third word. When the request number is 2 and the size is 6, the interpretation of bit 7 of a character attribute byte is returned in the third word. If this is 0, bit 7 of an attribute commands a blinking background. If it is 1, bit 7 of an attribute commands a high-intensity background color.

Discussion: See 10.4.1.6 through 10.4.1.8 **Include:** VIOCALLS
Related: *VioSetState*, *VioGetMode*, *VioGetFont*, *VioGetCurType*
Family API: available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:

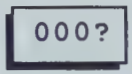
421 The request type must be 0, 1 or 2; the first register number must lie between zero and 15
436 The display handle must be zero
438 Structure length must agree with request type and with implied number of registers for request type 0

465 A detached process has no display access

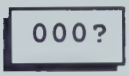
```
extern unsigned far pascal VIOGETSTATE(
                        struct VioState far *,
                        HANDLE);        /* must be zero */
#define VioGetState VIOGETSTATE

function VioGetState    (vars TheStuff : VioState;
                              ZeroHdl : Handle
                        ) : word; external;
```

VioModeUndo

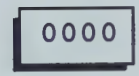
Cancel modewait watchdog




```
ownership option:
 1=give up ModeWait rights,
 0=reserve them
```




```
thread-end option:
 1=terminate waiting thread
 0=wake it up and send it a
   nonzero error code
```




```
handle of default display (must be zero)
```

If this process has set up a ModeWait watchdog via a call to *VioModeWait*, the watchdog is cancelled. If the second parameter is 1, the watchdog thread is terminated; if 0, the thread will receive control back from the *VioModeWait* call with a nonzero return code.

The first process in a screen group to issue *VioModeWait* is the owner of ModeWait rights for the screen group. The first parameter specifies what to do with that right. If it is 1, the process is finished with programming the display hardware and will not be setting up another watchdog; another process in the screen group can seize the right by issuing *VioModeWait*. If it is 0, this process reserves the right to set up another watchdog; another process will not be able to.

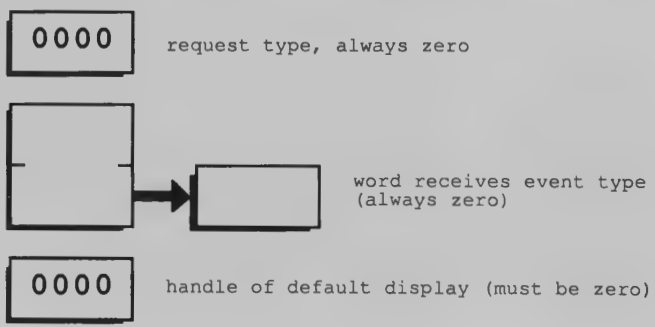
Discussion: See 10.5.5, 10.5.4 **Include:** VIOE
Related: *VioModeWait*
Family API: not available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:

427 No ModeWait thread has been established by this process

```
extern unsigned far pascal VIOMODEUNDO(
                        int, int,
                        HANDLE);        /* must be zero */
#define VioModeUndo VIOMODEUNDO

function VioModeUndo    (CedeOpt, KillOpt: integer;
                         ZeroHdl : Handle
                         ) : word; external;
```

VioModeWait

Suspend modewait watchdog



A ModeWait watchdog is an independent thread that is part of an application that programs the display hardware directly. The watchdog thread has the sole task of restoring the state of the display hardware following a pop-up display. The watchdog thread issues this call and is suspended in it. The call will return with a zero return code when a pop-up display has been shown and ended while this application's screen group was visible.

The watchdog should restore the display hardware registers to the state that its main thread expects, then reissue the call. There may be only one ModeWait watchdog per screen group. The first thread to issue the call becomes the owner of ModeWait rights for the screen group; other threads that issue the call will receive a nonzero return code. The call will also return a nonzero return code when the main thread calls *VioModeUndo* to terminate the watchdog.

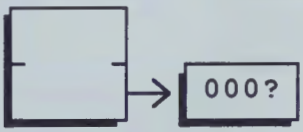
Discussion: *See* 10.5.5, 10.5.4 **Include:** VIDEO
Related: *VioModeUndo, VioSavRedrawWait*
Family API: not available

During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
422 Only one process per screen group may do this
423 Wait ended by a VioModeUndo call

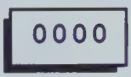
```
extern unsigned far pascal VIOMODEWAIT(
                        int, /* always 0 */
                        int far *, /* receives 0 */
                        HANDLE);        /* must be zero */
#define VioModeWait VIOMODEWAIT

function VioModeWait    (       AnyZero : integer; { must be zero }
                         vars   EvenTyp : integer; { receives zero }
                                ZeroHdl : Handle
                        ) : word; external;
```

VioPopup

Begin a pop-up display



wait option:
1=wait for popup screen group
0=return at once if it is in use



handle of default display (must be zero)

The pop-up display is a separate screen group comprising a text-mode display, a keyboard and a mouse. When in use, it preempts the screen group that is currently in the foreground. The operator cannot switch away from the pop-up display with the usual hot key and must interact only with it until it is ended by *VioEndPopup*.

This call requests the use of the pop-up display screen group. The first parameter indicates what to do if it is already in use. If it is zero, an error code will be returned; if 1, the current thread will be suspended until the pop-up screen group is free and then the call will take effect.

The pop-up display is initialized to 80x25 text mode with a screen cleared to blanks. A limited set of *Vio* calls may be used to write to the display, and keyboard and mouse calls may be used to get operator input.

Discussion: *See* 10.4.4 **Include:** VIOPOPUP
Related: *VioEndPopup*
Family API: not available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
406 The pop-up screen group is in use (and first parameter is zero)
430 This may not be done during a pop-up display

```
extern unsigned far pascal VIOPOPUP(
                        int far *, /* 0, or 1 to wait */
                        HANDLE);        /* must be zero */
#define VioPopup VIOPOPUP

function VioPopup       (vars WaitOpt : integer;
                             ZeroHdl : Handle
                        ) : word; external;
```

VioPrtSc

Copy screen to printer

0000	handle of default display (must be zero)

This call is used by the Session Manager to print the contents of the logical display buffer for the current screen group, in response to the operator's pressing the Shift-PrtSc key. The standard OS/2 code will reject a call from any other process. The call is documented here only because a replacement display subsystem registered with *VioRegister* might need to take over its function.

Discussion: *See* 10.6 **Include:** none
Related: *VioPrtScToggle*
Family API: available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
402 Only the Session Manager may do this

VioPrtScToggle

Invert state of printer-echo toggle

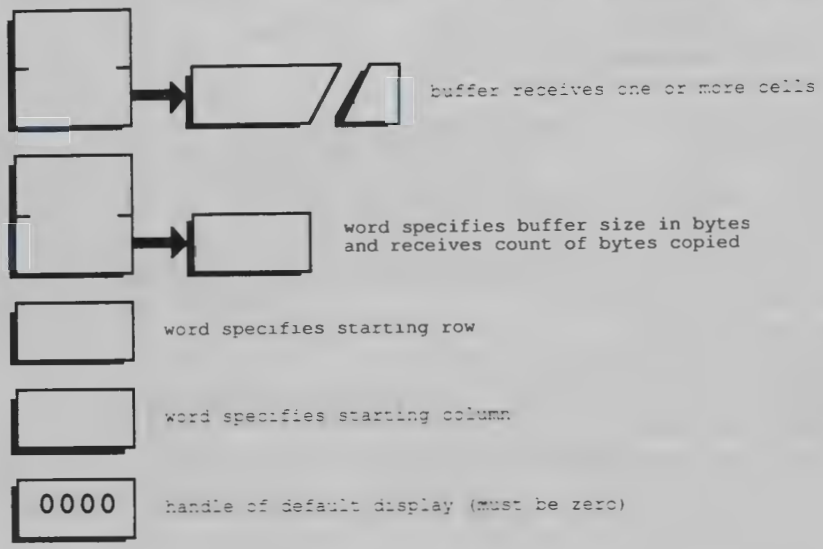
0000	handle of default display (must be zero)

This call is used by the session manager to invert the state of the printer-echo toggle in the current screen group, in response to the operator's pressing the Control-PrtSc key. The standard OS/2 code will reject a call from any other process. The call is documented here only because a replacement display subsystem registered with *VioRegister* might need to take over its function.

Discussion: *See* 10.6 **Include:** (no include file)
Related: *VioPrtSc*
Family API: available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
402 Only the Session Manager may do this

VioReadCellStr

Copy character cells from display buffer



Two-byte character cells from the logical display buffer for this screen group are copied into the supplied buffer. The first cell copied is the one at the specified row and column (the upper left corner of the screen is row zero, column zero; use *VioGetMode* to find out the screen dimensions).

Cells are copied continuously, row by row from the starting point, to the end of the screen or until the number of bytes specified by the second parameter is reached. The number of bytes copied is returned in the second parameter. If that is an odd number the last cell is incomplete. The logical cursor position is not affected by this call.

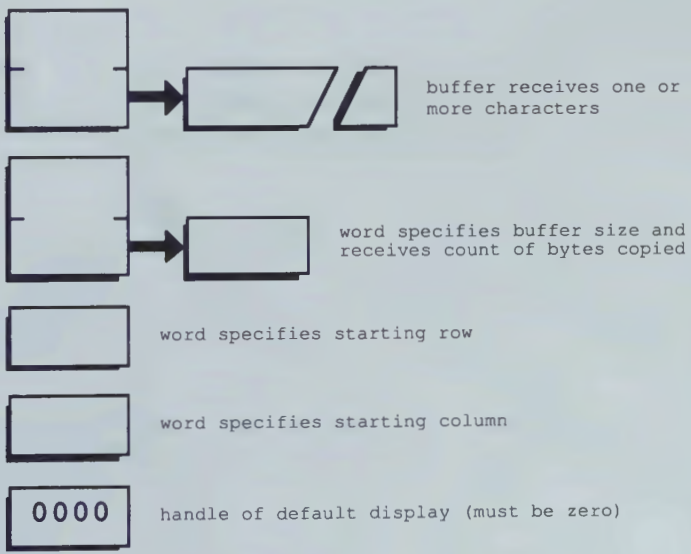
Discussion: *See* 10.4.2.1, 10.4.2.2, 10.4.2.4, **Include:** VIDEO
Related: *VioReadCharStr*, *VioWrtCellStr*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:

358 The row number must be between 0 and the screen height less 1
359 The column number must be between 0 and the screen width less 1
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOREADCELLSTR(
                        struct VioCell far *,
                        int far *,              /* size */
                        int, int,              /* row, col */
                        HANDLE);        /* must be zero */
#define VioReadCellStr VIOREADCELLSTR

function VioReadCellStr(vars FirstOne : VioCell; {e.g.
MyCellBuff[1]}
                        vars BufSize  : integer;
                        TheRow, TheCol : integer;
                        ZeroHdl : Handle
                        ) : word; external;
```

VioReadCharStr

Copy character bytes from display buffer



Successive characters from the logical display buffer for this screen group are copied into the supplied buffer. The first byte is the character at the specified row and column. (The upper left corner of the screen is row zero, column zero; use *VioGetMode* to find out the screen dimensions). Character bytes are copied continuously, row by row from the starting point, to the end of the screen or

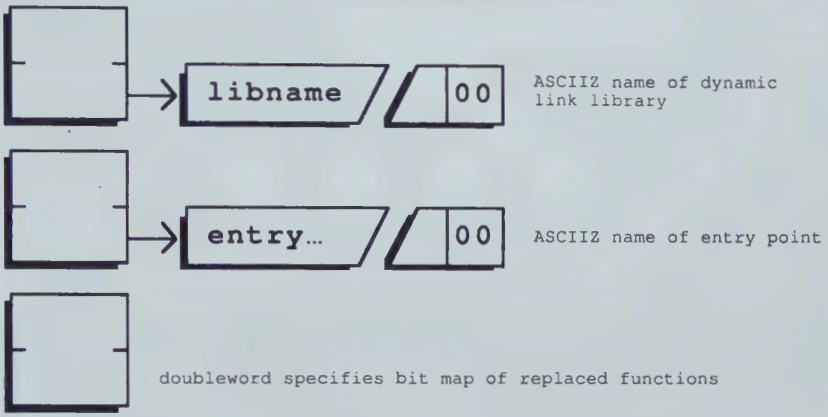
until the number of bytes specified in the second parameter is reached. The number of bytes copied is returned in the second parameter. The logical cursor position is not affected by this call.

Discussion: *See* 10.4.2.1, 10.4.2.2, 10.4.2.4 **Include:** VIDEO
Related: *VioReadCellStr, VioWrtCharStr*
Family API: available
During Pop-up: not available **Presentation Manager:** available
Errors to consider:
358 The row number must be between 0 and the screen height less 1
359 The column number must be between 0 and the screen width less 1
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOREADCHARSTR(
                        struct VioCell far *,
                        int far *,              /* size */
                        int, int,              /* row, col */
                        HANDLE);        /* must be zero */
#define VioReadCharStr VIOREADACHARSTR

function VioReadCharStr(vars FirstOne : char;   {e.g. MyBuff[1]}
                        vars BufSize  : integer;
                        TheRow, TheCol : integer;
                        ZeroHdl : Handle
                        ) : word; external;
```

VioRegister

Register replacement display subsystem



The dynamic link library specified is loaded as for *DosLoadModule*, and the entry point named is looked up in it as for *DosGetProcAddr*. If this is done successfully, the display functions indicated by 1-bits in the bit map are redirected so that, when one is called by any process in this screen group, control will pass instead to the entry point. See the discussion topic for the requirements on the designated entry point.

Table 10.6, p. 322 shows the relationship between replaced functions and bit numbers in the third parameter. Treat the third parameter as a 32-bit integer ("long" in C, "integer4" in Pascal) with bit 0 the least significant, bit 31 the most significant; in assembly language push bits 31-16 first, bits 15-0 second.

Only one display subsystem may be registered in a screen group. *VioDeregister* must be called to revoke the registration before another registration may be made. Most replaced functions are invoked only from processes in this screen group, and regardless of whether the screen group is visible or not. A replacement for *VioPopup*, however, will be invoked when any process in the system issues that call, but only when this screen group in the foreground.

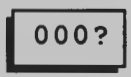
Discussion: *See* 10.6, 10.4.4
Include: VIDEO
Related: *VioDeregister*
Family API: not available
During Pop-up: not available **Presentation Manager**: not available

Errors to consider:
349 Invalid bit map (all zero?)
426 Only one VioRegister allowed per screen group

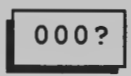
```
extern unsigned far pascal VIOREGISTER(
                        char far *,     /* libname */
                        char far *,     /* entry name */
                        long);          /* bitmap */
#define VioRegister VIOREGISTER

function VioRegister    (consts LibName, EntryPt : char; { e.g.
AZStr[1] }
                        TheMap : integer4
                        ) : word; external;
```

VioSavRedrawUndo

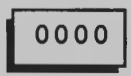
Cancel redraw watchdog




```
ownership option:
 1=give up Redraw rights,
 0=reserve them
```




```
thread-end option:
 1=terminate waiting thread
 0=wake it up and send it a
   nonzero error code
```




```
handle of default display (must be zero)
```

If this process has set up a Redraw watchdog via a call to *VioSavRedrawWait*, the watchdog is cancelled. If the second parameter is 1, the watchdog thread is terminated; if 0, the thread will receive control back from the *VioSavRedrawWait* call with a nonzero return code.

The first process in a screen group to issue *VioRedrawWait* is the owner of Redraw rights for the screen group. The first parameter specifies what to do with that right. If it is 1, the process is finished with programming the display hardware and will not be setting up another watchdog; another process in the screen group can seize the right by issuing *VioSavRedrawWait*. If it is 0, this process reserves the right to set up another watchdog.

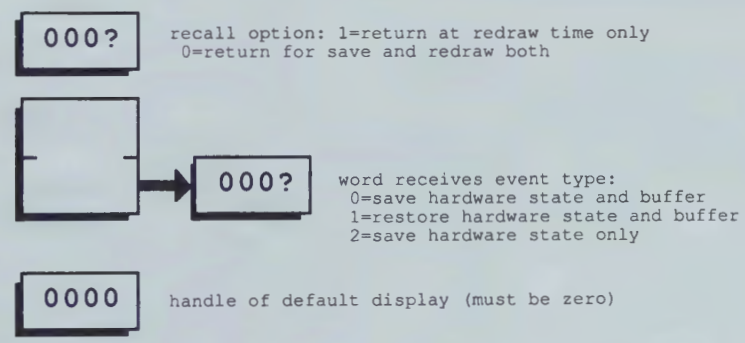
Discussion: *See* 10.5.4, 10.5.5 **Include:** VIDEO

Related: *VioSavRedrawWait, VioModeWait*
Family API: not available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
428 No Redraw watchdog has been established by this process

```
extern unsigned far pascal VIOREDRAWUNDO(
                        int,            /* cede ownership? */
                        int,            /* shoot the dog? */
                        HANDLE);        /* must be zero */
#define VioRedrawUndo VIOREDRAWUNDO

function VioRedrawUndo (CedeOpt, KillOpt : integer;
                        ZeroHdl : Handle
                        ) : word; external;
```

VioSavRedrawWait

Suspend redraw watchdog



A Redraw watchdog is an independent thread that is part of an application that programs the display hardware directly. The watchdog thread has two tasks: to redraw the display when the screen group is being made visible, and to save the state of the display when the system is switching away to another screen. If it is possible to redraw the display from program data without having saved it, the first parameter should be 1 and only the redraw event will be returned.

The watchdog thread issues this call and is suspended in it. The call will return with a zero return code when it is time to redraw or save the display. The reason for the return is set in the second parameter. If it is 1 for redraw, the code should

set up the hardware (any registers in the CGA, EGA or VGA on which it depends) and reconstruct the display in the physical display buffer. If the reason is 0 for save, the code should copy the hardware state and image data that can't be reconstructed at redraw time. If it is 2, save hardware state, only hardware registers need saving.

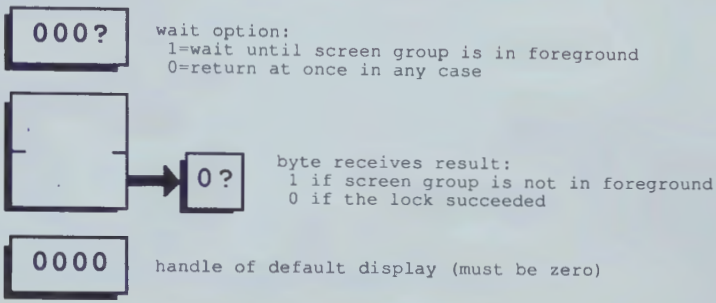
There may be only one Redraw watchdog per screen group. The first thread to issue the call becomes the owner of Redraw rights for the screen group; other threads that issue the call will receive a nonzero return code. The call will also return a nonzero return code when the main thread calls *VioSavRedrawUndo* to terminate the watchdog.

Discussion: *See* 10.5.4, 10.5.5 **Include:** VIDEO
Related: *VioSavRedrawUndo, VioModeWait*
Family API: not available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
422 Only one process in a screen group may do this
423 Wait ended by a *VioSavRedrawUndo* call

```
extern unsigned far pascal VIOSAVREDRAWWAIT(
                        int,            /* redraw only? */
                        int far *,      /* event type */
                        HANDLE);        /* must be zero */
#define VioSavRedrawWait VIOSAVREDRAWWAIT

function VioSavRedrawWait(     RecallOpt : integer;
                           vars EventType : integer;
                                  ZeroHdl : Handle
                        ) : word; external;
```

VioScrLock

Lock screen group in foreground



If the caller's screen group is not in the foreground (i.e. visible, connected to the real hardware), the first parameter is examined. If it is 1, the caller is suspended until the screen group is in the foreground; if zero, a value of 1 is set in the second parameter and the call ends.

When the caller's screen group is in the foreground it is locked there and zero is returned in the second parameter. In this state the system will not switch away immediately when the operator signals with a hot-key or a *VioPopup* call is made. The program may safely update the physical display buffer and graphics adapter hardware. It should do this as quickly as possible and then issue *VioScrUnlock*.

If a hot-key signal or *VioPopup* request occurs while the screen is locked, a timer is started. If at the end of an unspecified (but small) number of seconds the screen is still locked, all threads of the locking process are suspended and the screen switch is made regardless.

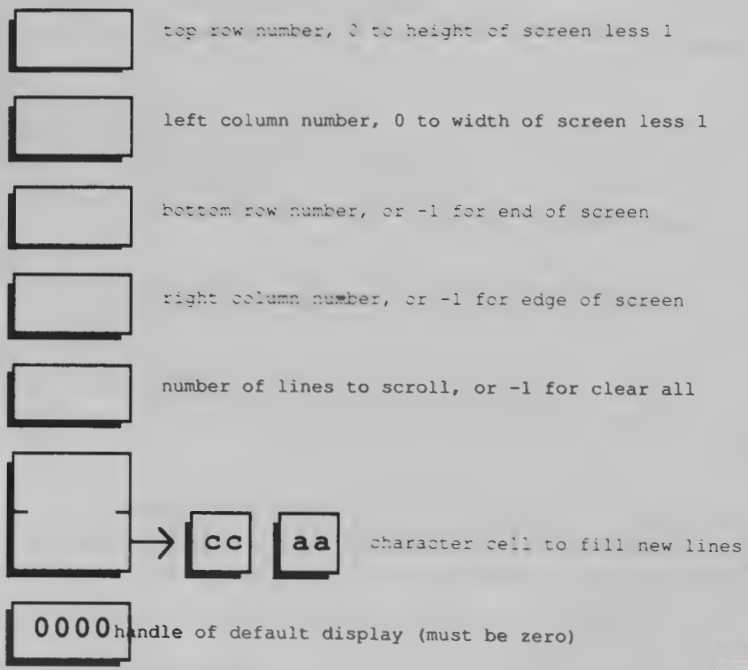
Discussion: *See* 10.5.2.2 **Include:** VIDEO
Related: *VioScrUnlock, VioSavRedrawWait, VioModeWait, VioGetPhysBuf*
Family API: call always succeeds with no wait
During Pop-up: not available **Presentation Manager**: not available
Errors to consider:
366 The wait option must be 0 or I
465 A detached process has no display access

```
extern unsigned far pascal VIOSCRLOCK(
                        int,              /* wait for fg? */
                        unsigned char far *, /* gets 0/1 */
```

```
                        HANDLE);        /* must be zero */
#define VioScrLock VIOSCRLOCK

function VioScrLock     (     WaitOpt : integer;
                        vars Outcome : byte;
                        ZeroHdl : Handle
                        ) : word; external;
```

VioScrollDn

Scroll a display region down



The lines in a rectangular region of the logical display buffer are scrolled down. New lines are added at the top of the region and filled with the specified cell value; old lines are lost at the bottom. If this screen group is visible, the physical display is updated.

The scrolled region is defined by its top left and bottom right corners, given as row and column numbers. (The top left corner of the screen is row 0, column 0;

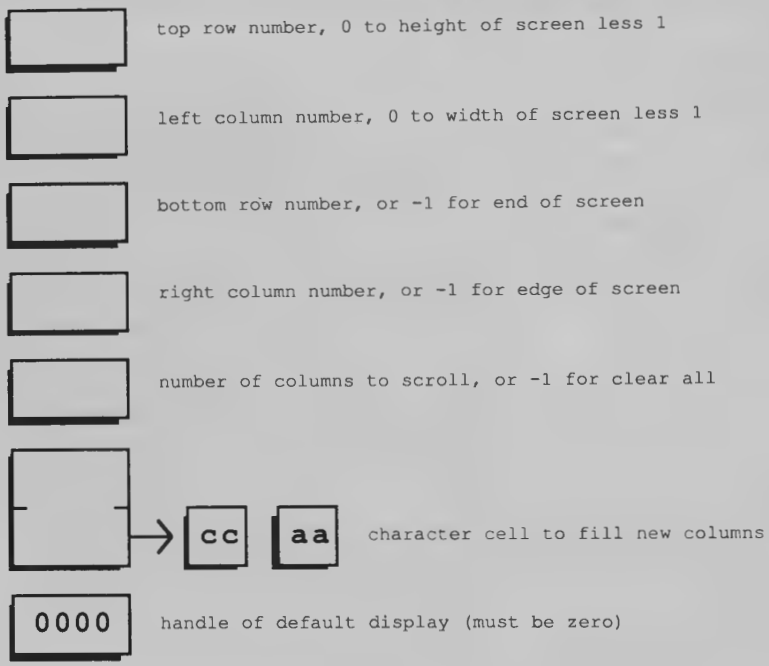
the dimensions of the screen can be found with *VioGetMode*.) A value of -1 for any row specifies the bottom row of the screen. A value of -1 for any column specifies the rightmost column of the screen. A value of -1 for the number of lines specifies all lines in the region, and will result in the region being cleared.

Discussion: *See* 10.4.2.5 **Include:** VIDEO
Related: *VioScrollUp, VioWrtCellStr, VioWrtCharStr*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
358 The bottom row number must be at least equal to top row number
359 The right column number must be at least equal to left column number
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOSCROLLDN(
                        int, int,       /* top row, left col */
                        int, int,       /* bot row, right col */
                        int,            /* number of lines */
                        struct VioCell far *, /* fill */
                        HANDLE);        /* must be zero */
#define VioScrollDn VIOSCROLLDN

function VioScrollDn    (TopRow, LeftCol,
                         BotRow, RightCol : integer;
                                 NumLines : integer;
                           consts TheCell : VioCell;
                                  ZeroHdl : Handle
                         ) : word; external;
```

VioScrollLf

Scroll a display region left



The columns of a rectangular region of the logical display buffer are scrolled left. New columns are added at the right side of the region and filled with the specified cell value; old columns are lost at the left. If this screen group is visible, the physical display is updated.

The scrolled region is defined by its top left and bottom right corners, given as row and column numbers. (The top left corner of the screen is row 0, column 0; the dimensions of the screen can be found with *VioGetMode*.) A value of -1 for any row specifies the bottom row of the screen. A value of -1 for any column specifies the rightmost column of the screen. A value of -1 for the number of columns specifies all columns in the region, and will result in the region being cleared.

Discussion: *See* 10.4.2.5 **Include:** VIDEO
Related: *VioScrollRt*
Family API: available

During Pop-up: available **Presentation Manager:** available

Errors to consider:

358 The bottom row number must be at least equal to top row number

359 The right column number must be at least equal to left column number

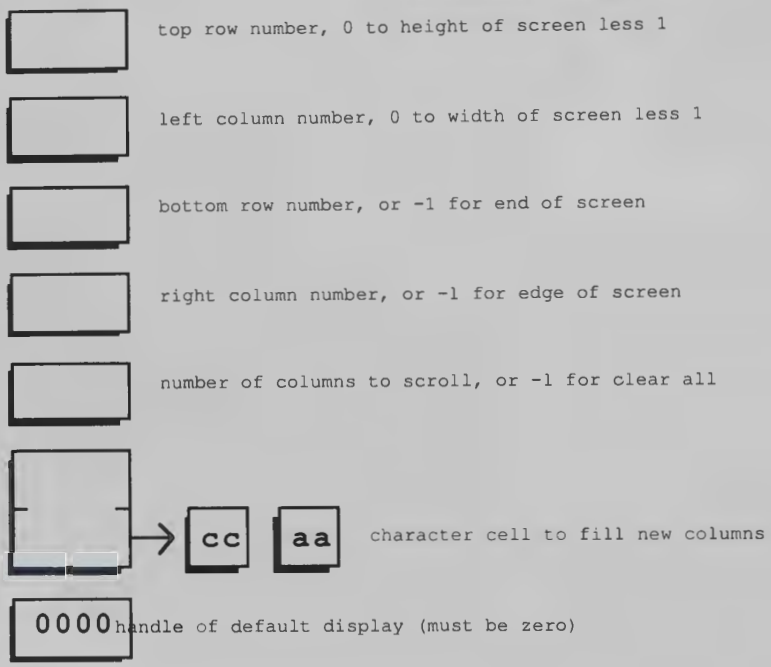
436 The display handle must be zero

465 A detached process has no display access

```
extern unsigned far pascal VIOSCROLLLF(
                        int, int,       /* top row, left col */
                        int, int,       /* bot row, right col */
                        int,            /* number of columns */
                        struct VioCell far *, /* fill */
                        HANDLE);        /* must be zero */
#define VioScrollLf VIOSCROLLLF

function VioScrollLf    (TopRow, LeftCol,
                         BotRow, RightCol : integer;
                                 NumCols  : integer;
                           consts TheCell : VioCell;
                                  ZeroHdl : Handle
                        ) : word; external;
```

VioScrollRt

Scroll a display region right



The columns of a rectangular region of the logical display buffer are scrolled right. New columns are added at the left side of the region and filled with the specified cell value; old columns are lost at the right. If this screen group is visible, the physical display is updated.

The scrolled region is defined by its top left and bottom right corners, given as row and column numbers. (The top left corner of the screen is row 0, column 0; the dimensions of the screen can be found with *VioGetMode*.) A value of -1 for any row specifies the bottom row of the screen. A value of -1 for any column specifies the rightmost column of the screen. A value of -1 for the number of columns specifies all columns in the region, and will result in the region being cleared.

Discussion: *See* 10.4.2.5 **Include:** VIDEO
Related: *VioScrollLf*
Family API: available

During Pop-up: available **Presentation Manager:** available

Errors to consider:

358 The bottom row number must be at least equal to top row number

359 The right column number must be at least equal to left column number

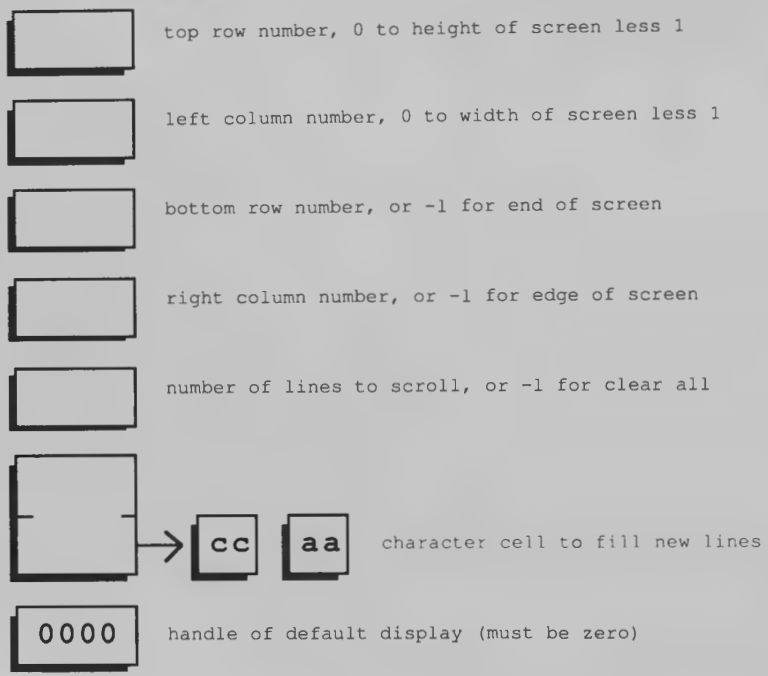
436 The display handle must be zero

465 A detached process has no display access

```
extern unsigned far pascal VIOSCROLLRT(
                        int, int,       /* top row, left col */
                        int, int,       /* bot row, right col */
                        int,            /* number of columns */
                        struct VioCell far *, /* fill */
                        HANDLE);        /* must be zero */
#define VioScrollRt VIOSCROLLRT

function VioScrollRt    (TopRow, LeftCol,
                         BotRow, RightCol : integer;
                                 NumCols  : integer;
                           consts TheCell : VioCell;
                                  ZeroHdl : Handle
                         ) : word; external;
```

VioScrollUp

Scroll a display region up



The lines in a rectangular region of the logical display buffer are scrolled up. New lines are added at the bottom of the region and filled with the specified cell value; old lines are lost at the top. If this screen group is visible, the physical display is updated.

The scrolled region is defined by its top left and bottom right corners, given as row and column numbers. (The top left corner of the screen is row 0, column 0; the dimensions of the screen can be found with *VioGetMode*.) A value of -1 for any row specifies the bottom row of the screen. A value of -1 for any column specifies the rightmost column of the screen. A value of -1 for the number of lines specifies all lines in the region, and will result in the region being cleared.

Discussion: *See* 10.4.2.5 **Include:** VIDEO
Related: *VioScrollDn, VioWrtCellStr, VioWrtCharStr*
Family API: available
During Pop-up: available **Presentation Manager:** available

Errors to consider:
358 The bottom row number must be at least equal to top row number
359 The right column number must be at least equal to left column number
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOSCROLLUP(
                        int, int,        /* top row, left col */
                        int, int,        /* bot row, right col */
                        int,             /* number of lines */
                        struct VioCell far *, /* fill */
                        HANDLE);         /* must be zero */
#define VioScrollUp VIOSCROLLUP

function VioScrollUp    (TopRow, LeftCol,
                         BotRow, RightCol : integer;
                                 NumLines : integer;
                           consts TheCell : VioCell;
                                  ZeroHdl : Handle
                         ) : word; external;
```

VioScrUnlock

Release locked screen group

(no parameters)

If the caller's screen group has been locked in the foreground with *VioScrLock*, it is unlocked; it becomes possible for the system to switch to a different screen group or to a pop-up display if that is called for.

Discussion: *See* 10.5.2.2 **Include:** VIOMGR
Related: *VioScrLock*
Family API: available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:
367 The screen group isn't locked

```
extern unsigned far pascal VIOSCRUNLOCK();
#define VioScrUnlock VIOSCRUNLOCK

function VioScrUnlock : word; external;
```

VioSetANSI

Set Ansi display support

000?	set value: 1=ANSI support on, 0=ANSI support off
0000	handle of default display (must be zero)

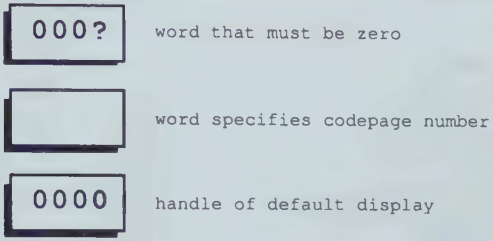
The support for ANSI escape sequences in the display output stream is turned on or off, according to the first parameter. The present state of ANSI support can be queried with *VioGetANSI*.

Discussion: *See* 10.3.4 **Include:** VIDEO
Related: *VioGetANSI*
Family API: not available
During Pop-up: not available **Presentation Manager:** available
Errors to consider:
421 The set value must be 0 or 1
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOSETANSI(
                        int,
                        HANDLE);        /* must be zero */
#define VioSetANSI VIOSETANSI

function VioSetANSI     (SetValue : integer;
                         ZeroHdl : Handle
                         ) : word; external;
```

VioSetCP

Select display codepage



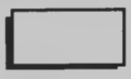
If the codepage number is zero, the display for this screen group is reset to use the default symbol translations built into the system. Otherwise the codepage must be one of those specified in the *country=* statement in the system configuration file, and the system must be able to find that codepage data in the file specified in the "devinfo ega..." configuration statement. If those things are true, the specified codepage is activated for the display for this screen group.Successful execution of this call clears a program-defined font set with *VioSet-Font.*

Discussion: *See* 10.4.1.5 **Include:** VIDEO
Related: *VioGetCP, KbdSetCP*
Family API: not available
During Pop-up: not available **Presentation Manager:** available
Errors to consider:
436 The display handle and first parameter must be zero
465 A detached process has no display access
469 The codepage number must be zero or defined in "country" statement
470 The hardware doesn't support loadable fonts

```
extern unsigned far pascal VIOSETCP(
                        int,            /* zero */
                        int,            /* id */
                        HANDLE);        /* must be zero */
#define VioSetCP VIOSETCP

function VioSetCP       (AnyZero : integer; { must be zero }
                         CPId    : integer;
                         ZeroHdl : Handle
                         ) : word; external;
```

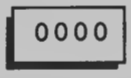
VioSetCurPos

Set cursor position in display buffer



row number, 0 tp screen height less 1

column number, 0 to screen width less 1



handle of default display (must be zero)

The position of the cursor in the display buffer for this screen group is set to the specified row and column. The upper left corner of the screen is row 0, column 0; the screen dimensions can be found with *VioGetMode*.

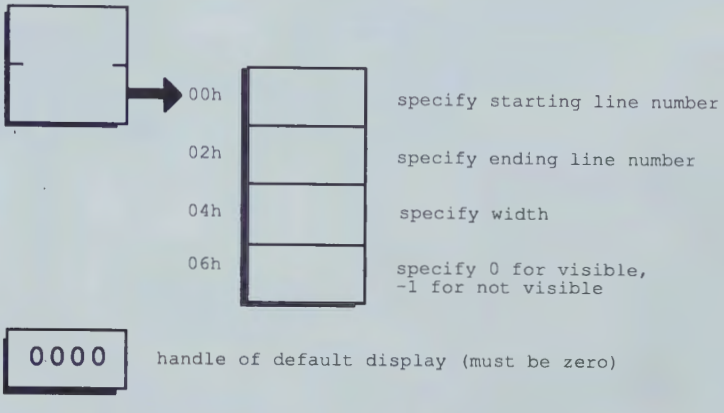
If the screen group is visible and in a text mode, the position of the cursor on the physical screen is updated. Output via *VioWrtTTY* will begin at the cursor position.

Discussion: *See* 10.4.1.3 **Include**: VIDEO
Related: *VioGetCurPos*, *VioSetCurType*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
358 The row number must be less than screen height
359 The column number must be less screen width
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOSETCURPOS(
                        int, int,
                        HANDLE);        /* must be zero */
#define VioSetCurPos VIOSETCURPOS

function VioSetCurPos  (TheRow, TheCol : integer;
                        ZeroHdl : Handle
                        ) : word; external;
```

VioSetCurType

Set appearance of cursor



The appearance of the text-mode cursor for this screen group is specified in a series of four words. The starting and ending line numbers describe the shape of the cursor. The units are lines of pels within a single character box, with line zero at the top of the box; hence the usual underscore cursor in a font with 8x14 characters starts on line 12 and ends on line 13. If the starting line exceeds the ending line the cursor "wraps around" the box from bottom to top. (The number of rows in a character box in the active font can be queried with *VioGetFont*.)

If the fourth field is zero the cursor will be visible on the screen; if it is nonzero (the preferred value is -1, 0ffffh) it will not be visible. The cursor width field is ignored; it should be specified as either zero or one.

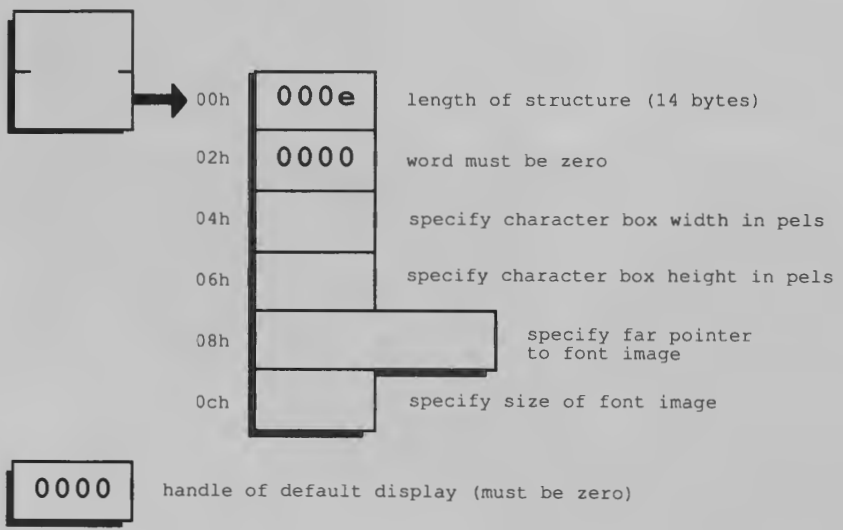
The recommended way of setting the cursor is to query this information with *VioGetCurType*, change the desired field(s) in place, and pass the modified structure to this call.

Discussion: *See* 10.4.1.4 **Include**: VIDEO
Related: *VioGetCurType*, *VioSetCurPos*, *VioGetMode*, *VioGetFont*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
356 The cursor width must be 0 or 1
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOSETCURTYPE(
                        struct VioCurType far *,
                        HANDLE);        /* must be zero */
#define VioSetCurType VIOSETCURTYPE

function VioSetCurType (consts TheStuff : VioCurType;
                        ZeroHdl : Handle
                        ) : word; external;
```

VioSetFont

Set display symbol font



A program-defined font, that is, display symbol map, is installed. The recommended way of setting an altered font is to query a font with *VioGetFont*, alter the data structure and font image it returns, and use those items as input to this call.

A program-defined font may only be set for hardware that supports software fonts (EGA and VGA). Use *VioConfig* to learn the hardware configuration.

A font image contains 256 bit maps, one to define a symbol for every possible byte value. Each bit map contains one byte for each row in a character box.

Where a byte contains a 1-bit the corresponding pel will be shown in the foreground color; a zero-bit pel will display in background color.

Successful use of this call resets the effect of *VioSetCP*. While a program-defined font is in effect, *VioGetCP* will return an error code. A call to *VioSetCP* with a parameter of zero may be used to reset the effect of this call.

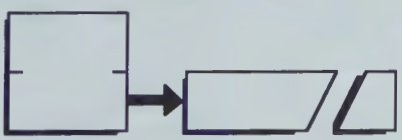
Discussion: *See* 10.4.1.5 **Include:** VIDEO
Related: *VioGetConfig, VioGetFont, VioGetCP, VioSetCP*
Family API: available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:

436 The display handle must be zero
438 Structure length must be 14; font space must be large enough for the entire font (256 times number of rows in a box); pel rows and columns must be reasonable
465 A detached process has no display access
467 Second word must be zero; character dimensions must be supported by the attached hardware

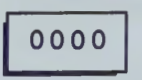
```
extern unsigned far pascal VIOSETFONT(
                        struct VioFont far *,
                        HANDLE);        /* must be zero */
#define VioSetFont VIOSETFONT

function VioSetFont     (consts TheStuff : VioFont;
                                ZeroHdl : Handle
                        ) : word; external;
```

VioSetMode

Modify display mode



display-mode structure, see figure VID.MDE



handle of default display (must be zero)

The display mode for this screen group is modified. The fields of the data structure are discussed in detail in the text. The recommended way of setting the mode

is to query it with *VioGetMode*, alter the fields of interest, and pass the altered structure to this call.

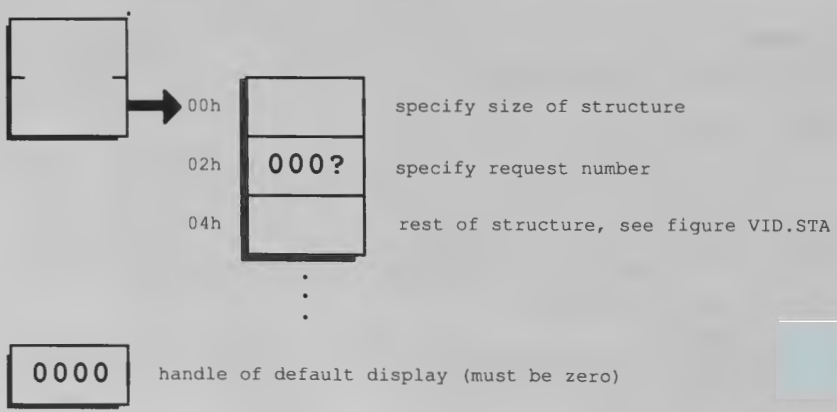
Under the Presentation Manager, only the number of text rows is set..

Discussion: *See* 10.4.1.2, Tables 10.3 and 10.4 **Include:** VIDEO
Related: *VioGetMode, VioGetConfig*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
355 That combination of numbers isn't a supported display mode
436 The display handle must be zero
465 A detached process has no display access
467 There's no font available to support that mode

```
extern unsigned far pascal VIOSETMODE(
                        struct VioMode far *,
                        HANDLE);        /* must be zero */
#define VioSetMode VIOSETMODE

function VioSetMode     (consts TheStuff : VioMode;
                                ZeroHdl : Handle
                        ) : word; external;
```

VioSetState

Modify display hardware settings



One of three hardware settings is modified depending on the number in the second field. In all cases, the size in the first word must be exact for the type of request and amount of data presented. The three formats of data structure and their contents are discussed under *VioGetState*.

The recommended way of setting the display state is to query it with *VioGetState*, alter the fields of interest, and pass the altered structure to this call.

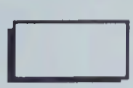
Discussion: *See* 10.4.1.6 through 10.4.1.8 **Include:** VIDEO
Related: *VioGetState*, *VioSetMode*, *VioSetCurType*
Family API: available
During Pop-up: not available **Presentation Manager:** not available
Errors to consider:

421 Request type must be 0, 1 or 2; first palette register and border color must be in 0-15; switch value must be 0 or 1
436 The display handle must be zero
438 The structure length word must match request type (and implied number of registers in request type 0)
465 A detached process has no display access

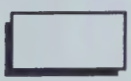
```
extern unsigned far pascal VIOSETSTATE(
                        struct VioState far *,
                        HANDLE);        /* must be zero */
#define VioSetState VIOSETSTATE

function VioSetState    (vars TheStuff : VioState;
                             ZeroHdl : Handle
                        ) : word; external;
```

VioShowBuf

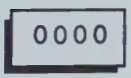
Notify alteration in display buffer



offset in buffer of first altered byte



length of alterations



handle of default display (must be zero)

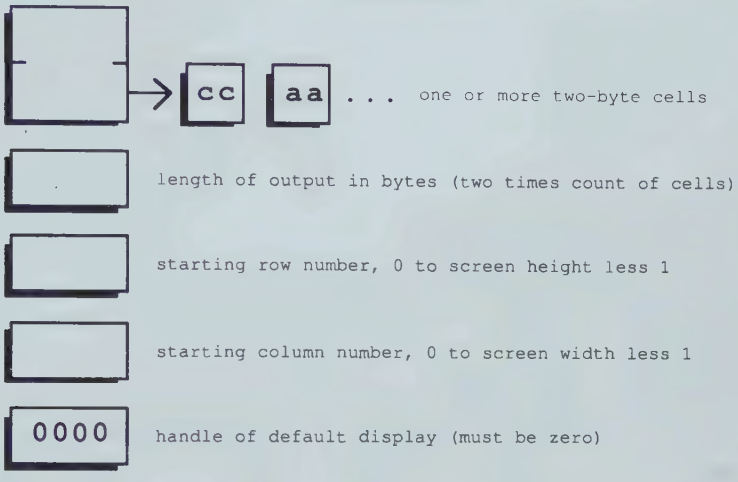
The system is notified that the program has altered the logical display buffer (whose address was obtained with *VioGetBuf*). If this screen group is presently visible, the altered portion of the display will immediately be copied into the physical display buffer and be visible on the screen. If the screen group is in the background nothing is done; the entire display buffer will be shown when the screen group is next switched to the foreground.

Discussion: *See* 10.4.3 **Include:** VIOCALLS
Related: *VioGetBuf*
Family API: available
During Pop-up: not available **Presentation Manager:** available
Errors to consider:
355 The display is in a graphics mode
430 This may not be done during a pop-up
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOSHOWBUF(
                        unsigned, /* offset */
                        unsigned, /* length */
                        HANDLE);
#define VioShowBuf VIOSHOWBUF

function VioShowBuf     (TheOffset, TheLength: word;
                         ZeroHdl : Handle
                         ) : word; external;
```

VioWrtCellStr

Write character cells to display buffer



One or more two-byte character cells are copied into the logical display buffer for this screen group, starting at the cell corresponding to a specified screen row and column. (The upper left corner of the screen is row 0, column 0; the screen dimensions can be learned with *VioGetMode*.)

Cells overlay the display continuously, row by row from the starting point, for the length of output or to the size of the screen. If the screen group is visible, the data are also copied into the physical display buffer and appear on the screen. The logical cursor position is not affected by this call.

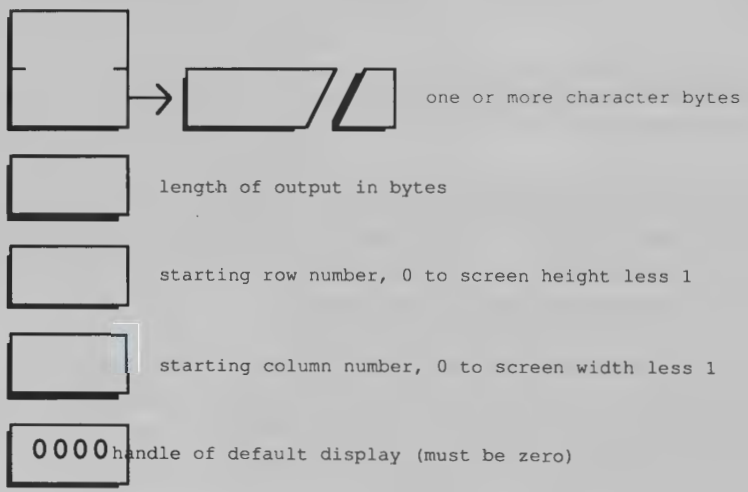
Discussion: *See* 10 4.2 **Include:** VIDEO
Related: other *VioWrt-* calls; *VioReadCellStr*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
358 The row number must be between 0 and the screen height less 1
359 The column number must be between 0 and the screen width less 1
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOWRTCELLSTR(
                        struct VioCell far *, /* cell string */
                        unsigned,       /* string length (bytes) */
```

```
                        int, int,       /* row, col */
                        HANDLE);
#define VioWrtCellStr VIOWRTCELLSTR

function VioWrtCellStr (consts TheCells : VioCell; { e.g. MyCells[1]
}
                                TheLength: word;
                                TheRow, TheCol : integer;
                                 ZeroHdl : Handle
                        ) : word; external;
```

VioWrtCharStr

Write characters to display buffer



One or more characters are copied into the logical display buffer for this screen group, starting at the position corresponding to a specified screen row and column. (The upper left corner of the screen is row 0, column 0; the screen dimensions can be learned with *VioGetMode*.)

Characters enter the display continuously, row by row from the starting point, for the length of output or to the size of the screen. If the screen group is visible, the updated display is copied into the physical display buffer and appears on the screen. The logical cursor position is not affected by this call.

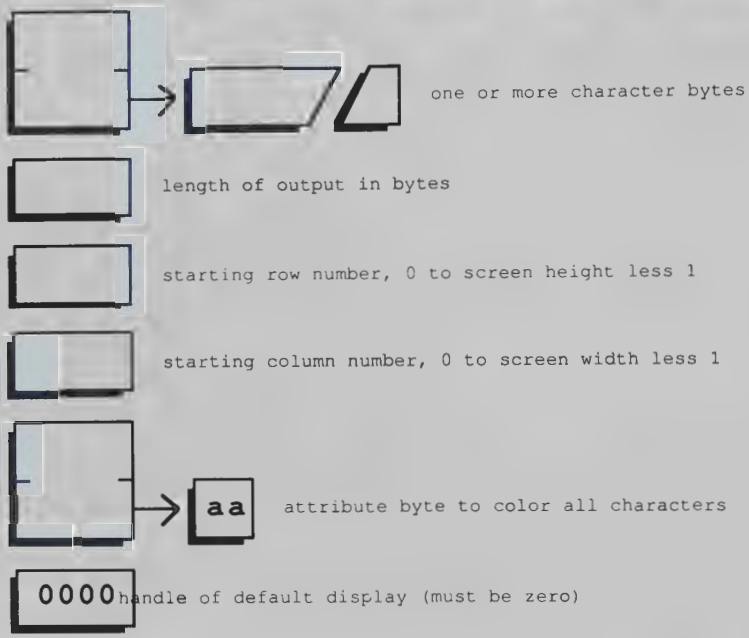
Discussion: *See* 10.4.2 **Include:** VIDEO

Related: other *VioWrt-* calls; *VioReadCharStr*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
358 The row number must be between 0 and the screen height less 1
359 The column number must be between 0 and the screen width less 1
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOWRTCHARSTR(
                        char far *,     /* cell string */
                        unsigned,       /* string length (bytes) */
                        int, int,       /* row, col */
                        HANDLE);
#define VioWrtCharStr VIOWRTCHARSTR

function VioWrtCharStr (consts TheChars : char; { e.g. MyStr[1] }
                               TheLength: word;
                               TheRow, TheCol : integer;
                               ZeroHdl : Handle
                        ) : word; external;
```

VioWrtCharStrAtt

Write characters to display buffer with attribute



One or more characters are copied into the logical display buffer for this screen group, starting at the position corresponding to a specified screen row and column. (The upper left corner of the screen is row 0, column 0; the screen dimensions can be learned with *VioGetMode*.)

Characters enter the display continuously, row by row from the starting point, for the length of output or to the size of the screen. The specified attribute is copied with each character. If the screen group is visible, the updated display is copied into the physical display buffer and appears on the screen. The logical cursor position is not affected by this call.

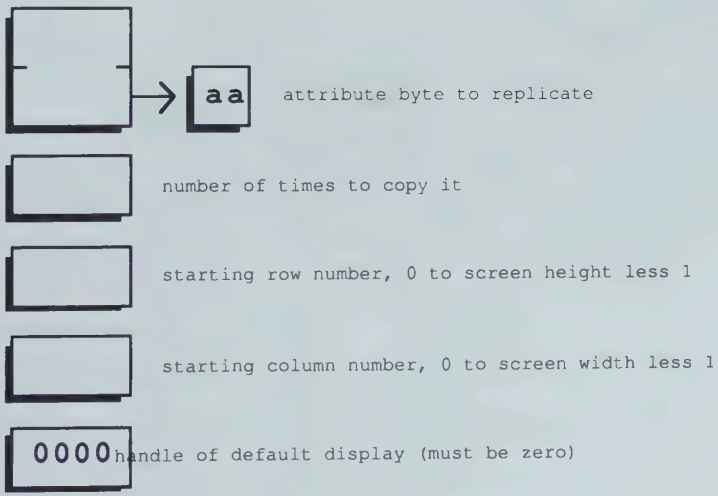
Discussion: *See* 10.4.2 **Include:** VIDEO
Related: *VioWrtCharStr*, *VioWrtNAttr*
Family API: available
During Pop-up: available **Presentation Manager**: available
Errors to consider:
358 The row number must be between 0 and the screen height less 1

359 The column number must be between 0 and the screen width less 1
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOWRTCHARSTRATT(
                        char far *,     /* string */
                        unsigned,       /* string length */
                        int, int,       /* row, col */
                        char far *,     /* attribute */
                        HANDLE);
#define VioWrtCharStrAtt VIOWRTCHARSTRATT

function VioWrtCharStrAtt(consts TheChars : char; { e.g. MyStr[1] }
                                  TheLength: word;
                                 TheRow, TheCol : integer;
                            consts TheAttr : byte;
                                 ZeroHdl : Handle
                          ) : word; external;
```

VioWrtNAttr

Replicate attributes in display buffer



One or more copies of an attribute are copied into the logical display buffer for this screen group, starting at the position corresponding to a specified screen row

and column. (The upper left corner of the screen is row 0, column 0; the screen dimensions can be learned with *VioGetMode*.)

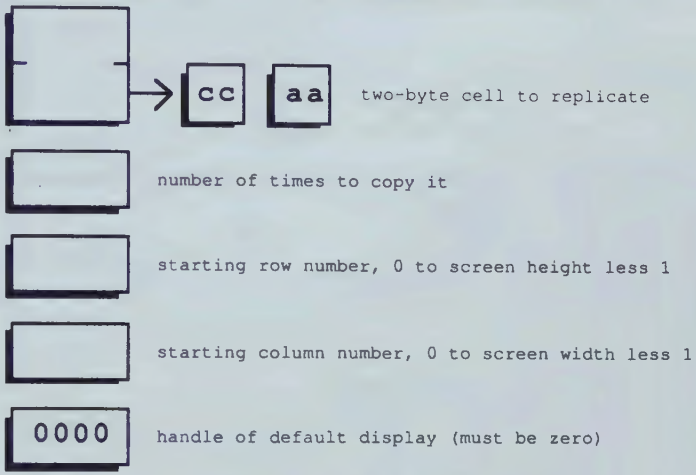
Attributes are applied across the display continuously, row by row from the starting point, for the specified count or to the size of the screen. If the screen group is visible, the updated display is copied into the physical display buffer and appears on the screen. The logical cursor position is not affected by this call.

Discussion: *See* 10.4.2 **Include:** VIDEO
Related: *VioWrtCharStrAtt, VioWrtCellStr*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
358 The row number must be between 0 and the screen height less 1
359 The column number must be between 0 and the screen width less 1
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOWRTNATTR(
                        unsigned char far *, /* attribute */
                        unsigned,       /* number of copies */
                        int, int,       /* row, col */
                        HANDLE);
#define VioWrtNAttr VIOWRTNATTR

function VioWrtNAttr   (consts TheAttr : byte;
                               TheCount: word;
                               TheRow, TheCol : integer;
                               ZeroHdl : Handle
                        ) : word; external;
```

VioWrtNCell

Replicate character cells in display buffer



One or more copies of a two-byte character cell are copied into the logical display buffer for this screen group, starting at the position corresponding to a specified screen row and column. (The upper left corner of the screen is row 0, column 0; the screen dimensions can be learned with *VioGetMode*.)

Copies of the cell overlay the display continuously, row by row from the starting point, for the specified count or to the size of the screen. If the screen group is visible, the updated display is copied into the physical display buffer and appears on the screen.The logical cursor position is not affected by this call.

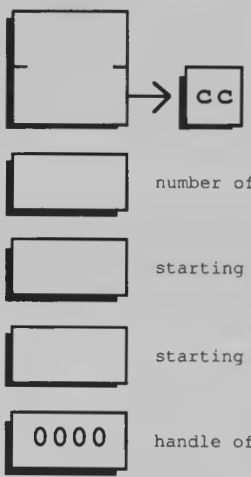
Discussion: *See* 10.4.2 **Include:** VIDEO
Related: *VioWrtCharStr, VioWrtCellStr, VioWrtNAttr*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
358 The row number must be between 0 and the screen height less 1
359 The column number must be between 0 and the screen width less 1
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOWRTNCELL(
                        struct VioCell far *, /* cell */
                        unsigned,       /* number of copies */
```

```
                        int, int,       /* row, col */
                        HANDLE);
#define VioWrtNCell VIOWRTNCELL

function VioWrtNCell   (consts TheCell : VioCell;
                               TheCount: word;
                               TheRow, TheCol : integer;
                               ZeroHdl : Handle
                        ) : word; external;
```

VioWrtNChar

Replicate characters in display buffer



One or more copies of a character byte are copied into the logical display buffer for this screen group, starting at the position corresponding to a specified screen row and column. (The upper left corner of the screen is row 0, column 0; the screen dimensions can be learned with *VioGetMode*.)

Characters enter the display continuously, row by row from the starting point, for the specified count or to the size of the screen. If the screen group is visible, the updated display is copied into the physical display buffer and appears on the screen. The logical cursor position is not affected by this call.

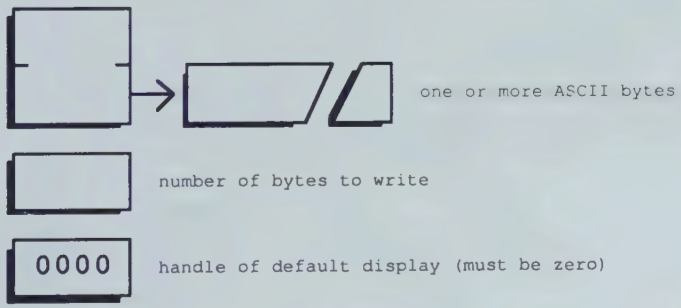
Discussion: *See* 10.4.2
Related: *VioWrtCharStr, VioWrtCharStrAtt*

Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
358 The row number must be between 0 and the screen height less 1
359 The column number must be between 0 and the screen width less 1
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOWRTNCHAR(
                        char far *,     /* character */
                        unsigned,       /* number of copies */
                        int, int,       /* row, col */
                        HANDLE);
#define VioWrtNChar VIOWRTNCHAR

function VioWrtNChar    (consts TheChar : char;
                                TheCount: word;
                                TheRow, TheCol : integer;
                                ZeroHdl : Handle
                        ) : word; external;
```

VioWrtTTY

Write ASCII stream to display



One or more ASCII codes are written to the logical display buffer starting at the current logical cursor position. Most byte values are placed in the display buffer as symbols, with the cursor position advancing to the last displayed byte. The following ASCII codes are interpreted as functions and not displayed:

- BEL (07h) sounds a beep.

- BSP (08h) moves the cursor left unless it is in column zero already.
- HT (09h) moves the cursor to the next eighth column right, moving down a row if necessary.
- LF (0ah) moves the cursor to the next lower row.
- CR (0dh) moves the cursor to column 0.

In addition, if ANSI escape sequences are enabled (see *VioGetANSI*), the ESC code (01bh) followed by a "[" character (05bh), is taken to be the start of an ANSI escape sequence. These two bytes and any following digits and semicolons are reserved and not displayed. The first nondigit character establishes the meaning of the sequence. If it is one of those shown in Table 10.1, p. 292 the ANSI function is performed. If not, display resumes with that unknown character.

Discussion: *See* 10.3.3, 10.3.4 **Include:** VIDEO
Related: *VioGetANSI*, *VioSetANSI*, *VioGetCurPos*, *VioSetCurPos*
Family API: available
During Pop-up: available **Presentation Manager:** available
Errors to consider:
436 The display handle must be zero
465 A detached process has no display access

```
extern unsigned far pascal VIOWRTTTY(
                        char far *,
                        int,
                        HANDLE);        /* must be zero */
#define VioWrtTTY VIOWRTTTY

function VioWrtTTY      (consts TheData : char; { e.g. AStr[1] }
                                TheSize : integer; { e.g.
ord(AStr[0]) }
                         ZeroHdl : Handle
                         ) : word; external;
```

Appendix A
Collected Data Structures

Here are the named data types used as parameters in the declarations of system calls in the Reference Section. These declarations were developed without reference to the source files distributed by IBM and/or Microsoft and as a result the names of types, structure fields, and often minor details of field declarations are different from those used in example programs from other sources. The *meaning* of the data types, when rendered into binary by a compiler, should never be different.

There are four divisions in this appendix. The first lists Typedefs and Defined constants for C. The second lists C structure declarations. The third lists Pascal Constants and the fourth, Pascal Types.

Within these sections, objects are ordered alphabetically by name. This sequence is for ease of reference by you; it is clearly not the best order for input to a compiler. The include files in the Disk Appendix are ordered differently.

A.1 Typedefs and Defined Constants for C

```
typedef void far * ADSMEM;
typedef void (far * ADSPROC)();
typedef unsigned ATTRWORD;
#define ATTNRM 0x0000
#define ATTRO  0x0001
#define ATTHID 0x0002
#define ATTSYS 0x0004
#define ATTVOL 0x0008
#define ATTDIR 0x0010
#define ATTARC 0x0020
typedef unsigned DATEWORD;
typedef long FILEOFFSET;
#define FP_FRONT 0
#define FP_BACK 2
#define FP_RELATIVE 1
```

```
#define FNAMELEN 13 /* max ASCIIZ filename.ext */
#define FPATHLEN 81 /* max ASCIIZ path string */
typedef unsigned HANDLE;
/* options for KbdCharIn, KbdStringIn */
#define KBDWAIT 0
#define KBDNOWAIT 1
/* Define MONBUFSIZE larger before this include if you wish */
#ifndef MONBUFSIZE
#define MONBUFSIZE 256 /* recommended size */
#endif
/* the data capacity of a monitor buffer is 18 bytes less */
#define MONBUFDATA (MONBUFSIZE-18)
#define NILADS ((void far *)0L)
#define NULLRANGE (struct LOCKRANGE far *)0l
typedef unsigned OPENFLAG;
/* add the following to make an OPENFLAG */
#define OF_OLD_FAIL 0
#define OF_OLD_OPEN 1
#define OF_OLD_TRUNC 2
#define OF_NEW_FAIL 0
#define OF_NEW_MAKE 16
typedef unsigned OPENMODE;
/* add the following to make an OPENMODE */
#define OM_ACC_RO 0
#define OM_ACC_WO 1
#define OM_ACC_RW 2
#define OM_INHERIT 128
#define OM_DENY_RW 16
#define OM_DENY_WO 32
#define OM_DENY_RO 48
#define OM_DENY_NONE 64
#define OM_RETURN_ERRS 0x2000
#define OM_WRITE_THRU 0x4000
#define OM_DASD 0x8000
typedef unsigned OPTIONSET;
typedef unsigned PROCESSID;
typedef unsigned SELECTOR;
typedef unsigned long SEMAPHORE;
typedef unsigned long far *SEMHANDLE;
/* values for Shift State in KeyRecord */
#define SH_RIGHT_SHIFT  0x0001
#define SH_LEFT_SHIFT   0x0002
#define SH_ANY_CTL      0x0004
#define SH_ANY_ALT      0x0008
#define SH_SCROLOCK     0x0010
#define SH_NUMLOCK      0x0020
#define SH_CAPSLOCK     0x0040
#define SH_INSERT       0x0080
#define SH_LEFT_CTL     0x0100
#define SH_LEFT_ALT     0x0200
#define SH_RIGHT_CTL    0x0400
#define SH_RIGHT_ALT    0x0800
#define SH_SCROLOCK_KEY 0x1000
#define SH_NUMLOCK_KEY  0x2000
#define SH_CAPSLOCK_KEY 0x4000
#define SH_SYSREQ_KEY   0x8000
```

```
typedef unsigned THREADID;
typedef long TIMEINTERVAL;
typedef unsigned TIMEWORD;
```

A.2 Structures for C

```
union ADS {
        ADSMEM ptr;
        struct { unsigned R,S; } wds;
        };
struct CTRYCPID {
/* used in DosCaseMap, GetCollate, GetCtryInfo, GetDBCSEv */
        int country, codepage;
                };
struct CTRYINFO { /* returned by DosGetCtryInfo */
        int     country;        /* country code */
        int     codepage;       /* codepage for print & casemap */
        int     dateform;       /* 0=mdy, 1=dmy, 2=ymd */
        char    moneymark[5];   /* ASCIIZ currency, eg "$" */
        char    thousandsep[2]; /* ASCIIZ thousands sep, eg. "," */
        char    decimalsep[2];  /* ASCIIZ decimal sep, eg. "." */
        char    datesep[2];     /* ASCIIZ date sep, eg. "/" */
        char    timesep[2];     /* ASCIIZ time sep, eg. ":" */
   unsigned char moneyform;     /* money format flags: */
#define MONEY_DECIMAL 0x04      /* use moneymark for decimalsep */
#define MONEY_AFTER 0x01        /* mark after amount if true */
#define MONEY_SPACE 0x02        /* space between mark and amount */
   unsigned char moneydec;      /* number of decimal places in money */
   unsigned char timeform;      /* time format: */
#define TIME_24 0x01            /* 23:00 if true, 11:00p if false */
        long    reserve2;       /* left over from MSDOS */
        char    itemsep[2];     /* ASCIIZ list separator eg. "," */
        int     reserve3[5];    /* reserved */
                };
struct DATETIME { /* structure returned by DosGetDateTime */
        unsigned char hours;    /* hour 0..23 */
        unsigned char minutes;  /* minute 0..59 */
        unsigned char seconds;  /* second 0..59 */
        unsigned char hundredths;/*  0..99      */
        unsigned char day_of_month; /* day 1..31 */
        unsigned char month;    /* month 1..12 */
        unsigned int  year;     /* year 1987..1999 */
                 int  timezone; /* minutes from GMT (PLUS if WEST) */
        unsigned char day_of_week; /* Sunday == 0 */
                };
struct FileDates { /* file date/time info used in several calls */
        DATEWORD create_date;
        TIMEWORD create_time;
        DATEWORD access_date;
        TIMEWORD access_time;
        DATEWORD write_date;
        TIMEWORD write_time;
                };
struct FileFind { /* file info from FindFirst/Next */
```

```
        struct FileInfo fi; /* e.g FF.fi.fd.write_date */
        char name_len; /* 0 to 12 */
        char fname[41]; /* variable */
                };
struct FileInfo { /* file info from QFileInfo and FindFirst/Next */
        struct FileDates fd;
        long end_of_data;
        long allocation;
        ATTRWORD attribute;
                };
struct FSInfo1 {
        char    system_ID[4];
        long    secs_per_unit;
        long    total_units;
        long    free_units;
        int     bytes_per_sec;
        };
struct FSInfo2 {
        DATEWORD label_date;
        TIMEWORD label_time;
        struct FSVolLabel L;
        };
union FSInfo {
        struct FSInfo1 F1;
        struct FSInfo2 F2;
                };
struct FSVolLabel {
        char    length;
        char    label[12];
        };
struct GLOBALSEG { /* one structure returned by DosGetInfoSeg */
        unsigned long clocksecs;/* time since 1-1-70 in seconds */
        unsigned long clockmils;/* time since bootup in millisex */
        unsigned char hours;    /* hour 0..23 */
        unsigned char minutes;  /* minute 0..59 */
        unsigned char seconds;  /* second 0..59 */
        unsigned char hundredths;/*  0..99      */
        unsigned int  timezone; /* minutes -/+ GMT */
        unsigned int  timerunit;/* timer precision in .0001 secs */
        unsigned char day_of_month; /* day 1..31 */
        unsigned char month;    /* month 1..12 */
        unsigned int  year;     /* year 1987..1999 */
        unsigned char day_of_week; /* Sunday == 0 */
        unsigned char vermajor; /* OS/2 version */
        unsigned char verminor; /* ditto */
        unsigned char revletter;/* revision letter */
        unsigned char fgsgnum;  /* foreground screen group (not nec. us) */
        unsigned char mxsgnum;  /* max screen groups */
        unsigned char hugeshift;/* huge shift */
        unsigned char protonly; /* 1 if no dos box allowed */
        unsigned int  fgpid;    /* current FG proc id (not nec. us) */
        unsigned char dynavar;  /* 1 if dynamic scheduler variations */
        unsigned char maxwait;  /* config maxwait= value (sec) */
        unsigned int  minslice; /* minimum time slice (ms) */
        unsigned int  maxslice; /* maximum time slice (ms) */
        unsigned int  bootdrive;/* boot drive number */
```

```
        unsigned char tracemap[32];   /* trace types enabled */
        };
struct KeyBufLen { /* parameter to KBDSTRINGIN */
                int buflen, tmotlen;
                        };
struct KeyRecord { /* received from KbdCharIn and KbdPeek */
        unsigned char key;
        unsigned char scan;
#define NLS_ONSPOT 0x20
#define NLS_FINAL 0x40
#define NLS_INTERIM 0x80
        unsigned char nls_stat;
        unsigned char nls_shft;
        unsigned int shift;
        long timer;
                };
struct KeyStatus { /* structure for KbdGet/SetStatus */
        int size; /* must be sizeof(struct KEYSTATUS) */
#define KSECHOON   0x0001
#define KSECHOOFF  0x0002
#define KSECHOBITS (KSECHOON|KSECHOOFF)
#define KSRAWMODE  0x0004
#define KSCOOKEDMODE 0x0008
#define KSMODEBITS (KSRAWMODE|KSCOOKEDMODE)
#define KS2BYTECR  0x0080
#define KSREPORTSHIFT 0x0100
        OPTIONSET mask;
        unsigned int turn;
        unsigned char nls_stat, junk;
        OPTIONSET shift;
                };
struct KeyXlate { /* exchanged with KbdXlate */
        struct KeyRecord kr;
        OPTIONSET kddflags;
        unsigned xflag;
        unsigned xstate;
        unsigned xrsvrd;
                };
struct LOCALSEG { /* other structure from DosGetInfoSeg */
        unsigned int  ourpid;   /* my proc. id */
        unsigned int  momspid;  /* my parent's process id */
        unsigned char ourpriority;  /* my thread's priority */
        unsigned char ourclass;  /* ..and dispatching class */
        unsigned int  ourtid;   /* my thread's id */
        unsigned int  ourSGnum; /* my screen group */
        unsigned int  oursubSG; /* my subscreen group or screen subgroup */
        unsigned int  now_in_fg;/* nonzero if ourpid == fgpid */
        };
struct LockRange {
                FILEOFFSET start, length;
                };
struct MCOLLISION { /* collision area specified to MouRemovePtr */
        int row;        /* upper row */
        int col;        /* left column */
        int height;     /* number of rows */
        int width;      /* number of columns */
```

```
                };
struct MEVENT { /* Mouse event returned by MouReadEventQue */
        unsigned event; /* see MEV defines above */
        long time;      /* millisecond timestamp */
        int  row;       /* mouse row */
        int  col;       /* mouse column */
                };
struct MonBuffer {
        int size; /* must be sizeof(struct MonBuffer) on MonReg call */
        unsigned header[8]; /* modified during MonReg call */
        char therest[MONBUFDATA];
                };

struct MPTRPOS { /* pointer position used in MouGet/SetPtrPos */
        int row;
        int col;
                };
struct MPTRDEF { /* pointer definition used in MouGet/SetPtrShape */
        unsigned total_size; /* size of pointer in bytes */
        int colsize;    /* width in pixels, 1 for text mode */
        int rowsize;    /* height in pixels, 1 for text mode */
        int hotcol;     /* hot-spot pixel column, 1 for text */
        int hotrow;     /* hot-spot pixel row, 1 for text */
                };
struct MQSTATUS { /* Mouse Queue Status returned by MouGetNumQueEl */
        int now_in_queue;
        int max_in_queue;
                };
struct MSCALE { /* scale factor used in MouGet/SetScaleFact */
        int row_scale;
        int col_scale;
                };
struct MUXWAITLIST {
                        int count;
                        struct {
                                int skipword;
                                SEMHANDLE handle;
                                } list[16];
                    };
struct PIDDESC { /* returned by DosReadQueue, DosPeekQueue */
        PROCESSID writerPID; /* PID of process writing Q item */
        unsigned  itemdesc;  /* descriptive word from WriteQueue */
        };
struct PROCIDSET { /* returned by DosGetPID */
        PROCESSID myPID;
        THREADID myTID;
        PROCESSID parentPID;
                };
structure STARTDATA {
                int sd_size; /* sizeof(struct STARTDATA) */
                int sd_related;
                int sd_fgbg;
                int sd_trace; /* "session trace indicator" */
                char far * sd_title;
                char far * sd_progfile;
                char far * sd_parms;
```

```
                char far * sd_queue;
                        }
struct STATUSDATA {
        int std_size; /* sizeof(struct STATUSDATA) */
        int std_select, std_bind;
                        };
struct VioCell { /* a character cell */
        unsigned char symbol, attrib;
                };
struct VioConfig { /* used in VioGetConfig, VioSetConfig */
        int reclen;     /* must be sizeof(struct VioConfig) */
        int adapter, display;
        long ramsize;
                };
struct VioCurType { /* used in VioGetCurType, VioSetCurType */
        int starts, ends, width, attrib;
                };
struct VioFont { /* used in VioGetFont, VioSetFont */
        int reclen;     /* must be sizeof(struct VioFont) */
        int fontype;    /* 0 for active, 1 for ROM */
        int cwidth, cheight;
        char far * fontaddr;
        int fontlen;
                };
struct VioMode { /* used in VioGetMode, VioSetMode */
        int reclen;     /* must be sizeof(struct VioMode) */
#define VioNotMono 1
#define VioGraphOn 2
#define VioBurstOff 4
        unsigned char modebits;
        unsigned char colorexp;
        int txtcols, txtrows;
        int pelcols, pelrows;
                };
struct VioPhysBuf { /* used in VioGetPhysBuf */
        long unsigned physad; /* note: not an ADS type */
        long    physlen;
        SELECTOR sels[2];
                };
struct VioState { /* used in VioGetState, VioSetState */
        int reclen;     /* must be 38, 6 or 6 */
        int reqno;      /* must be 0, 1 or 2 */
        union { /* variant record, case of reqno: */
                int firstreg;   /* 0 : first reg to set */
                int border;     /* 1 : border color */
                int attbit7;    /* 2 : char attr bit 7 usage */
                } u;
        int palregs[16];        /* 0 only: palette regs to set */
                };
```

A.3 Pascal Constants

```
Const
    FNameLen = 13; { max ASCIIZ filename.ext string }
    FPathLen = 81; { max file path string }
    FP_Front = 0;
    FP_Back = 2;
    FP_Relative = 1;
    { keystroke device driver flag values }
    KDDBreak     = 16#0040; { packet is a break signal }
    KDDSecondary = 16#0080; { secondary decode }
    KDDMultimake = 16#0100; { multimake signal (?) }
    KDDAccented  = 16#0200; { packet is accented character }
    KDDSignal    = 16#003f; { mask to extract signal code }
    { options to KbdCharIn, KbdStringIn }
    KeyWait = 0;
    KeyNoWait = 1;
    { keystroke monitor flag-byte values }
    KMonOpen =  16#01;      { packet represents an open }
    KMonClose = 16#02;      { packet represents a close }
    KMonFlush = 16#04;      { packet requests flush/reset }
    MonBufSize = 256;       { recommended monitor buffer size }
    MsgMaxIns = 9;
    NilADS = ADSMEM(0,0);
    NilADSPROC = ADSPROC(0,0);
    { values in NLS_Stat bytes }
    NLS_OnSpot = 16#20;
    NLS_Final = 16#40;
    NLS_Interim = 16#80;
    NullRange = ADSRange(0,0);
    { add the following to make an OpenFLAG for DosOpen }
    OF_Old_Fail = 0;        { if file exists, fail }
    OF_Old_Open = 1;        { if file exists, open }
    OF_Old_Trunc= 2;        { if file exists, set file size }
    OF_New_Fail = 0;        { if file is new, fail }
    OF_New_Make = 16;       { if file is new, make it to size }
    { add the following to make an OpenMODE for DosOpen }
    OM_Acc_RO   = 0;        { open for reading only }
    OM_Acc_WO   = 1;        { open for writing only }
    OM_Acc_RW   = 2;        { open for both }
    OM_Inherit  = 128;      { child may inherit this handle }
    OM_Deny_RW  = 16;       { deny all other opens }
    OM_Deny_WO  = 32;       { deny only writers, readers ok }
    OM_Deny_RO  = 48;       { deny only readers }
    OM_Deny_None= 64;       { deny nobody, share with all }
    OM_Return_Errs = 16#2000;{ return physical errors }
    OM_Write_Thru = 16#4000;{ do not buffer output }
    OM_DASD = 16#8000; { DASD Open }
```

A.4 Pascal Types

```
Type
    ADSRange = ADSMEM;
    AttrWord = set of FileAttrs;
    CtryCPID = record
        { used in DosCaseMap, GetCollate, GetCtryInfo, GetDBCSEv }
        country, codepage : integer
    end;
    CtryInfo = packed record { returned by DosGetCtryInfo }
        country      [00]: integer; { country code }
        codepage     [02]: integer; { codepage for print & casemap }
        dateform     [04]: integer; { 0=mdy, 1=dmy, 2=ymd }
        moneymark    [06]: array[1..5] of char;{ ASCIIZ currency mark, eg "$" }
        thousandsep  [11]: array[1..2] of char;{ ASCIIZ thousand sep, eg. "," }
        decimalsep   [13]: array[1..2] of char;{ ASCIIZ decimal sep, eg. "." }
        datesep      [15]: array[1..2] of char;{ ASCIIZ date sep, eg. "/" }
        timesep      [17]: array[1..2] of char;{ ASCIIZ time sep, eg. ":" }
        moneyform    [19]: byte;    { money format, see Consts }
        moneydec     [20]: byte;    { number of decimal places in money }
        timeform     [21]: byte;    { see Time24 constant above }
        reserve2     [22]: integer4;{ left over from MSDOS }
        listsep      [26]: array[1..2] of char;{ ASCIIZ list sep eg "," }
        reserve3     [28]: array[1..5] of integer
    end;
    DateTime = record { structure used by DosGet/SetDateTime }
        hours   : byte; { hours 0..23 }
        minutes : byte; { minutes 0..59 }
        seconds : byte; { seconds 0..59 }
        hundredths : byte; { fractions of seconds 0..99 }
        DayOfMonth : byte; { day 1..31 }
        month   : byte; { month 1..12 }
        year : 1987..1999; { year }
        timezone: -720..720; { minutes from GMT (PLUS == WEST) }
        DayOfWeek : byte; { 0..6, Sunday == 0 }
    end;
    DateWord = word;
    DaySpan = 1..31;
    FileAttrs = (AttRO,AttHid,AttSys,AttVol,AttDir,AttArc);
    FileDates = record { date/time info used in several calls }
        create_date : DateWord;
        create_time : TimeWord;
        access_date : DateWord;
        access_time : TimeWord;
        write_date  : DateWord;
        write_time  : TimeWord
                end;
    FileFind = record { file info returned by Find calls }
        fi : FileInfo; { for example, FF.fi.fd.write_date }
        fname : Lstring(41)
                end;
    FileInfo = record { file info from QFileInfo and Find calls }
        fd : FileDates;
        EndOfData : integer4;
        Allocation : integer4;
        Attribute : AttrWord
```

```
            end;
FileOffset = integer4;
FSInfo1 = record
    SystemID : packed array[1..4] of char;
    SecsPerUnit : integer4;
    TotalUnits : integer4;
    FreeUnits : integer4;
    BytesPerSec : integer
            end;
FSInfo2 = record
    LabelDate : DateWord;
    LabelTime : TimeWord;
    LabelText : FSVolLabel;
            end;
FSInfo = record
    case integer of
    1 : (F1 : FSInfo1);
    2 : (F2 : FSInfo2)
            end;
FSVolLabel = string(12);
GlobalSeg = record { 1st structure returned by DosGetInfoSeg }
    clocksecs: integer4;{ time since 1-1-70 in seconds }
    clockmils: integer4;{ time since bootup in millisex }
    hours    : 1..23;   { hour 1..24 }
    minutes  : 0..59;   { minute 0..59 }
    seconds  : 0..59;   { second 0..59 }
    hundredths:0..99;   { fraction 0..99 }
    timezone : integer; { minutes -/+ GMT }
    timerunit: word;    { timer precision in .0001 secs }
    DayOfMonth:1..31;   { day 1..31 }
    month:     1..12;   { month 1..12 }
    year:  1987..1999;  { year 1987..1999 }
    DayOfWeek: 0..7;    { Sunday = 0 }
    VerMajor:  byte;    { OS/2 version }
    VerMinor:  byte;    { ditto }
    RevLetter: char;    { revision letter }
    FGSGnum:   byte;    { screen group now in FG (not nec. us) }
    MaxSGnum:  byte;    { max screen groups }
    HugeShift: byte;    { huge shift }
    ProtOnly:  byte;    { 1 if no dos box allowed }
    FGPID:     word;    { current FG proc id (not nec. us) }
    Dynavar:   byte;    { 1 if dynamic scheduler variations }
    MaxWait:   byte;    { config maxwait= value (sec) }
    MinSlice:  word;    { minimum time slice (ms) }
    MaxSlice:  word;    { maximum time slice (ms) }
    BootDrive: word;    { boot drive number, C: = 0003 }
    Tracemap: set of byte; { trace types enabled }
end;
Handle = word;
HourSpan = 1..24;
KeyBufLen = record { parameter to KbdStringIn }
    BufLen, TmOtLen : integer
            end;
KShifts = (ShiftRight, ShiftLeft, CtlAny, AltAny,
           ScrollLock, NumLock, CapsLock, Insert,
           CtlLeft, AltLeft, CtlRight, AltRight,
```

```
          ScrollKey, NumKey, CapsKey, SysReqKey);
KeyRecord = record { received from KbdXlate in, KbdPeek }
    Key      : BYTE;
    Scan     : BYTE;
    NLS_Stat: BYTE;
    NLS_Shft: BYTE;
    Shift    : packed set of KShifts;
    timer    : integer4
           end;
KeyStatFlags = (EchoOn, EchoOff, RawMode, CookedMode, SetShift,
    ,              SetNLS, SetTurn, Turn2Bytes );
KeyStatus = record { KbdSetStatus structure }
    KSLen    : integer; { must be 10 }
    KSMask   : packed set of KeyStatFlags;
    KSTurn   : WORD; { usually 13 for ord(Enter) }
    KSNLS    : BYTE;
    rsvdK    : BYTE; {skip}
    KSShift  : packed set of KShifts
           end;
KeyXlate = record { input to KbdXlate }
    KR       : KeyRecord;
    KDDFlags: WORD;
    XFlag    : WORD; { nonzero on finished xlation }
    XState   : WORD; { xlation state code }
    XRsrvd   : WORD
           end;
LocalSeg = record { other structure from DosGetInfoSeg }
    OurPID:     word;    { my  proc. id }
    MomsPID:    word;    { my parent's process id }
    OurPriority:word;    { my thread's priority }
    OurTID:     word;    { my thread's id }
    OurSGnum:   word;    { my screen group }
    OurSubSG:   word;    { my subscreen group or screen subgroup }
    NowInFG:    word;    { nonzero if you can see my output }
end;
LockRange = record
    start, length : FileOffset
           end;
MCollision = record { collision area specified to MouRemovePtr }
    bot_row,                { lower-right row }
    right_col,              { lower-right column }
    top_row,                { upper-left row }
    left_col: integer       { upper-left column }
           end;
MEvent = record { Mouse event returned by MouReadEventQue }
    event : word;           { see MEV defines above }
    time  : integer4;       { millisecond timestamp }
    row,                    { mouse row }
    col   : integer         { mouse column }
           end;
MinuteSpan = 1..59;
MonBuffer = record      { 2 each given to MonReg }
    BufSize : integer; { must be MonBufSize when MonReg done }
    TheRest : array[1..MonBufSize-2] of byte
           end;
MonthSpan = 1..12;
```

```
MPtrPos = record { pointer position used in MouGet/SetPtrPos }
    row : integer;
    col : integer
          end;
MPtrDef = record { pointer definition used in MouGet/SetPtrShape }
    total_size : word;      { size of pointer image in bytes }
    colsize,                { width in pixels, 1 for text mode }
    rowsize,                { height in pixels, 1 for text mode }
    hotcol,                 { hot-spot pixel column, 1 for text }
    hotrow : integer        { hot-spot pixel row, 1 for text }
          end;
MQStatus = record { Mouse Queue Status returned by MouGetNumQueEl }
    now_in_queue : integer;
    max_in_queue : integer
          end;
MScale = record { scale factor used in MouGet/SetScaleFact }
    row_scale : integer;
    col_scale : integer
          end;
MsgINSList = array[1..MsgMaxIns] of ADS of char;
MuxWaitList = record
    count : integer;
    list : array[1..16] of record
                   skipword : word;
                   handle : SemHandle
                   end
          end;
OpenFlag = word;{ see OF_ constants }
OpenMode = word;{ see OM_ constants }
OptionSet = packed set of 0..15;
PIDDesc = record { returned by DosReadQueue, DosPeekQueue }
    WriterPID : ProcessID;  { PID of process writing Q item }
    ItemDesc  : word        { descriptive word from WriteQueue }
end;
ProcessID = word;
ProcIDSet = record { items returned by DosGetPID }
    MyPID : word;
    MyTID : word;
    ParentPID : word
end;
SecondSpan = MinuteSpan;
Selector = word;
Semaphore = integer4;
SemHandle = ADS of Semaphore;
SigActions = (SigDefault, SigIgnore, SigSetHandler,
              SigReject, SigAcknowledge);
SigTypes   = (SigIntr, Sig2Skipped, SigBreak,
               SigFlagA, SigFlagB, SigFlagC);
StartData = record
    sdSize : integer;
    sdRelated : integer;
    sdFgBg : integer;
    sdTrace : integer; { "session trace indicator" }
    sdTitle,
    sdProgfile,
    sdParms,
```

```
    sdQueue : ADS of char
            end;
  StatusData = record
    stdSize, { 6 }
    stdSelect,
    stdBind : integer
            end;
  TimeInterval = integer4;
  TimeWord = word;
  ThreadID = word;
  VioCell = record { text-mode character cell }
     'Symbol, Attrib : byte;
            end;
  VioConfig = record { used in VioGetConfig, VioSetConfig }
    RecLen  : integer; { must be sizeof(VioConfig) }
    Adapter : integer;
    Display : integer;
    RamSize : integer4
            end;
  VioCurType = record { used in VioGetCurType, VioSetCurType }
    Starts  : integer;
    Ends    : integer;
    Width   : integer;
    Attrib  : integer
            end;
  VioFont = record { used in VioGetFont, VioSetFont }
    RecLen  : integer; { must be sizeof(VioFont) }
    FonType : integer; { 0 for active, 1 for ROM }
    Cwidth  : integer;
    Cheight : integer;
    FontAddr: ADSMEM;
    FontLen : integer
            end;
  VioMode = record { used in VioGetMode, VioSetMode }
    RecLen  : integer; { must be sizeof(VioMode) }
    ModeBits: BYTE; { see Vio- consts }
    ColorExp: BYTE;
    TxtCols, TxtRows : integer;
    TelCols, PelRows : integer
            end;
 VioPhysBuf = record { used in VioGetPhysBuf }
    PhysAdr : integer4; { note: not an ADS type }
    PhysLen : integer4; { typically 16000 }
    Sels    : array[0..1] of Selector
            end;
  VioState = record { used in VioGetState, VioSetState }
    RecLen : integer; { must be exact size of record Variant! }
    case ReqNo : integer of { must be 0, 1 or 2 }
    0 : (   FirstReg : integer;
            RegList : array[0..15] of integer; );
    1 : (   Border : integer );
    2 : (   AttBit7 : integer );
            end;
  YearSpan = 1980..2107;
```

More Software Tools from M&T Books

Programming Languages

C Chest and Other C Treasures from *Dr. Dobb's Journal*
Edited by Allen Holub
Item #40-2 $24.95 (book)
Item #49-6 $39.95 (book/disk)
This comprehensive anthology contains the popular "C Chest" columns from *Dr. Dobb's Journal of Software Tools*, along with the lively philosophical and practical discussions they inspired, in addition to other information-packed articles by C experts. The software in the book is also available on disk with full source code. MS-DOS format.

Turbo C: The Art of Advanced Program Design, Optimization, and Debugging
Stephen R. Davis
Item #38-0 $24.95 (book)
Item #45-3 $39.95 (book/disk)
Overflowing with example programs, this book fully describes the techniques necessary to skillfully program, optimize, and debug in Turbo C. All programs are also available on disk with full source code. MS-DOS format.

C Programming for MIDI
Jim Conger
Item #86-0 $22.95 (book)
Item #90-9 $37.95 (book/disk)
For musicians and programmers alike, here is the source that will help you write programs for music applications. The author begins by outlining the features of MIDI (Musical Instrument Digital Interface) and its support of real-time access to musical devices. An introduction to C programming fundamentals as they relate

to MIDI is also provided. The author fully demonstrates these concepts with two MIDI applications: a patch librarian and a simple sequencer.

Dr. Dobb's Toolbook of C
Editors of *Dr. Dobb's Journal*
Item #89303-615-3 $29.95
From *Dr. Dobb's* and Brady Communications, this book contains a comprehensive library of valuable C code. *Dr. Dobb's* most popular articles on C are updated and reprinted here, along with new C programming tools. Also included is a complete C compiler, an assembler, text processing programs, and more!

Dr. Dobb's Toolbook of Forth
Edited by Marlin Ouverson
Item #10-0 $22.95 (book)
Item #57-7 $39.95 (book/disk)
This comprehensive collection of useful Forth programs and tutorials contains expanded versions of *DDJ*'s best Forth articles and other material, including practical code and in-depth discussions of advanced Forth topics. The screens in the book are also available on disk as ASCII files in the following formats: MS/PC-DOS, Apple II, Macintosh, or CP/M: Osborne or 8" SS/SD.

Dr. Dobb's Toolbook of Forth, Volume II
Editors of *Dr. Dobb's Journal*
Item #41-0 $29.95 (book)
Item #51-8 $45.95 (book/disk)
This complete anthology of Forth programming techniques and developments picks up where the Toolbook of Forth, First Edition left off. Included are the best articles on Forth from *Dr. Dobb's Journal of Software Tools,* along with the latest material from other Forth experts. The screens in the book are available on disk as ASCII files in the following formats: MS-DOS, Macintosh, and CP/M: Osborne or 8" SS/SD.

The New BASICs: Programming Techniques and Library Development
Namir Clement Shammas
Item #37-2 $24.95 (book)
Item #43-7 $39.95 (book/disk)
This book will orient the advanced programmer to the syntax and programming features of The New BASICs, including Turbo BASIC 1.0, QuickBASIC 3.0, and True BASIC 2.0. You'll learn the details of implementing subroutines,

functions, and libraries to permit more structured coding. Programs and subroutines are available on disk with full source code. MS-DOS format.

The Turbo Pascal Toolbook
Edited by Namir Clement Shammas
Item #25-9 $25.95 (book)
Item #61-5 $45.95 (book/disk)
This book contains routines and sample programs to make your programming easier and more powerful. You'll find an extensive library of low-level routines; external sorting and searching tools; window management; artificial intelligence techniques; mathematical expression parsers, including two routines that convert mathematical expressions into RPN tokens; and a smart statistical regression model finder. More than 800K of source code is available on disk for MS-DOS systems.

Programming Tools and Utilities

The Small-C Handbook
James E. Hendrix
Item #8359-7012-4 $17.95 (book)
Item #67-4 $37.90 (book and CP/M disk)
The Small-C Handbook with MS/PC-DOS Addendum and Disk
James E. Hendrix
Item #76-3 $42.90 (book/addendum/MS-DOS disk)
Also from *DDJ* and Brady Communications, the handbook is a valuable companion to the Small-C compiler, described below. The book explains the language and the compiler, and contains entire source listings of the compiler and its library of arithmetic and logical routines.

Small-C Compiler
James E. Hendrix
Item #01-1 $19.95
Like a home study course in compiler design, the *Small-C Compiler* and *The Small-C Handbook* provide all you need to learn how compilers are constructed, as well as teaching the C language at its most fundamental level. Full source code is included on disk in both CP/M and MS/PC-DOS versions.

A Small C Compiler: Language, Theory, and Design
James E. Hendrix
Item #88-7 $23.95
A full presentation of the design and theory of the Small C compiler (including source code) and programming language. The author has implemented many features in this compiler that make it an excellent example for learning basic compiler theory. Some of these features are: recursive dissent parsing, one-pass compilation, and the generation of assembly language. Here is a look into a real compiler with the opportunity for hands-on experience in designing one.

Small-Windows: A Library of Windowing Functions for the C Language
James E. Hendrix
Item #35-X $29.95
Small-Windows is a complete windowing library for C. The package includes video functions, menu functions, window functions, and more. The package is available for MS-DOS systems for the following compilers: Microsoft C Version 4.0, Small-C, and Lattice C. Documentation and full C source code is included.

Small Tools: Programs for Text Processing
James E. Hendrix
Item #78-X $29.95 (manual/disk)
This package of text-processing programs written in Small-C is designed to perform specific, modular functions on text files. Source code is included. Small Tools is available in both CP/M and MS/PC-DOS versions and includes complete documentation.

Small-Mac: An Assembler for Small-C
James E. Hendrix
Item #77-1 $29.95 (manual/disk)
Small-Mac is a macro assembler designed to stress simplicity, portability, adaptability, and educational value. Small-Mac is available for CP/M systems only and includes source code on disk with complete documentation.

NR: An Implementation of the Unix NROFF Word Processor
Allen Holub
Item #33-X $29.95
NR is a text formatter that is written in C and compatible with Unix's NROFF. *NR* comes configured for any Diablo-compatible printer, as well as Hewlett Packard's ThinkJet and LaserJet. Both the ready-to-use program and full source code are included. For PC compatibles.

Statistical Toolbox for Turbo Pascal
Namir Clement Shammas
Item #22-4 $69.95 (manuals/disks)
Two statistical packages in one! A library disk and reference manual that includes statistical distribution functions, random number generation, basic descriptive statistics, parametric and nonparametric statistical testing, bivariate linear regression, and multiple and polynomial regression. The demonstration disk and manual incorporate these library routines into a fully functioning statistical program. For IBM PCs and compatibles.

Turbo Advantage
Lauer and Wallwitz
Item #26-7 $49.95
A library of more than 200 routines, with source code sample programs and documentation. Routines are organized and documented under the following categories: bit manipulation, file management, MS-DOS support, string operations, arithmetic calculations, data compression, differential equations, Fourier analysis and synthesis, and much more! For MS/PC-DOS systems.

Turbo Advantage: Complex
Lauer and Wallwitz
Item #27-5 $89.95
This library provides the Turbo Pascal code for digital filters, boundary-value solutions, vector and matrix calculations with complex integers and variables, Fourier transforms, and calculations of convolution and correlation functions. Some of the *Turbo Advantage: Complex* routines are most effectively used with Turbo Advantage. Source code and documentation included.

Turbo Advantage: Display
Lauer and Wallwitz
Item #28-3 $69.95
Turbo Advantage: Display includes an easy-to-use form processor and thirty Turbo Pascal procedures and functions to facilitate linking created forms to your program. Full source code and documentation are included. Some of the *Turbo Advantage* routines are necessary to compile *Turbo Advantage: Display*.

Time and Task Management with dBASE III
Timothy Berry
Item #09-7 $49.95 (manual/MS-DOS disk)
Like an accounting system for time and tasks, this package helps users organize hours, budgets, activities, and resources. Providing both a useful time-

management system and a library of dBASE III code and macros, this package has practical as well as educational value. To be used with dBASE III. Source code and documentation is included. MS-DOS disk format.

Sales Management with dBASE III
Timothy Berry
Item #15-1 $49.95 (manual/MS-DOS disk)
Sales management works with dBASE III to provide a powerful information system that will help you to keep track of clients, names, addresses, follow-ups, pending dates, and account data. This system organizes all the day-to-day activities of selling and includes program files, format files, report files, index files, and data bases. Documentation and full source code is included.

Operating Systems

On Command: Writing a Unix-Like Shell for MS-DOS
Allen Holub
Item #29-1 $39.95
Learn how to write shells applicable to MS-DOS, as well as to most other programming environments. This book and disk include a full description of a Unix-like shell, complete C source code, a thorough discussion of low-level DOS interfacing, and significant examples of C programming at the system level. All source code is included on disk.

/util: A Unix-Like Utility Package for MS-DOS
Allen Holub
Item #12-7 $29.95
This collection of utilities is intended to be accessed through SH but can be used separately. It contains programs and subroutines that, when coupled with SH, create a fully functional Unix-like environment. The package includes a disk with full C source code and documentation in a Unix-style manual.

Taming MS-DOS, Second Edition
Thom Hogan
Item #87-9 $19.95
Item #92-5 $34.95
Described by reviewers as "small in size, large on content," and "fun." The second edition promises to be just as readable and is updated to cover MS-DOS 3.3. Some of the more perplexing elements of MS-DOS are succinctly described here with time-saving tricks to help customize any MS-DOS system. Each trick is easily implemented into your existing tools and for programmers, Hogan

includes many complete source code files that provide very useful utilities. All source code is written in BASIC.

Program Interfacing to MS-DOS
William G. Wong
Item #34-8 $29.95
Program Interfacing to MS-DOS will orient any experienced programmer to the MS-DOS environment. The package includes a ten-part manual with sample program files and a detailed description of how to build device drivers, along with the device driver for a memory disk and a character device driver on disk with macro assembly source code.

Tele Operating System Toolkit
Ken Berry
This task-scheduling algorithm drives the Tele Operating System and is composed of several components. When integrated, they form an independent operating system for any 8086-based machine. Tele has also been designed for compatibility with MS-DOS, UNIX, and the MOSI standard.

SK: THE SYSTEM KERNEL
Item #30-5 $49.95 (manual/disk)
The System Kernel contains an initialization module, general-purpose utility functions, and a real-time task management system. The kernel provides MS-DOS applications with multitasking capabilities. The System Kernel is required by all other components. All source code is included on disk in MS-DOS format.

DS: WINDOW DISPLAY
Item #32-1 $39.95 (manual/disk)
This component contains BIOS level drivers for a memory-mapped display, window management support and communication coordination between the operator and tasks in a multitasking environment. All source code is included on disk in MS-DOS format.

FS: THE FILE SYSTEM
Item #65-8 $39.95 (manual/disk)
The File System supports MS-DOS disk file structures and serial communication channels. All source code is included on disk in MS-DOS format.

XS: THE INDEX SYSTEM
Item #66-6 $39.95 (manual/disk)
The Index System implements a tree-structured free-form database. All source code is included on disk in MS-DOS format.

UNIX Programming on the 80286/80386
Alan Deikman
Item #83-6 $24.95 (book)
Item #91-9 $39.95 (book/disk)
A complete professional level tutorial and reference for programming UNIX and Xenix on 80286/80386 based computers. Succinct coverage of the UNIX program environment, UNIX file system, shells, utilities, and C programming under UNIX are covered. The author also delves into the development of device drivers; some examples of these are video displays, tape cartridges, terminals, and networks.

The Programmer's Essential OS/2 Handbook
David E. Cortesi
Item #82-8 $24.95 (book)
Item #89-5 $39.95 (book/disk)
Here is a resource no developer can afford to be without! Cortesi succinctly organizes the many features of OS/2 into related topics and illuminates their uses. Multiple indexes and a web of cross referencing in the margins provide easy access to all OS/2 topic areas. Equal support for Pascal and C programmers is provided. *The* essential reference for programmers developing in the OS/2 environment.

Dr. Dobb's Toolbook of 80286, 80386 Programming
Editors of *Dr. Dobb's Journal*
Item #42-9 $24.95 (book)
Item #53-4 $39.95 (book/disk)
This toolbook is a comprehensive discussion on the powerful 80X86 family of microprocessors. The editors of *Dr. Dobb's Journal of Software Tools* have gathered their best articles, updated and expanded them, and added new material to create this valuable resource for all 80X86 programmers. All programs are available on disk with full source code.

Chips

Dr. Dobb's Z80 Toolbook
David E. Cortesi
Item #07-0 $25.00 (book)
Item #55-0 $40.00 (book/disk)
This book contains everything users need to write their own Z80 assembly-language programs, including a method of designing programs and coding them in assembly language and a complete, integrated toolkit of subroutines. All the software in the book is available on disk in the following formats: 8" SS/SD, Apple, Osborne, or Kaypro.

Dr. Dobb's Toolbook of 68000 Programming
Editors of *Dr. Dobb's Journal*
Item #13-216649-6 $29.95 (book)
Item #75-5 $49.95 (book/disk)
From *DDJ* and Brady Communications, this collection of practical programming tips and tools for the 68000 family contains the best 68000 articles reprinted from *DDJ* along with much new material. The book contains many useful applications and examples. The software in the book is also available on disk in the following formats: MS/PC-DOS, Macintosh, CP/M 8", Osborne, Amiga, and Atari 520ST.

X68000 Cross Assembler
Brian R. Anderson
Item #71-2 $25.00
This manual and disk contain an executable version of the 68000 Cross Assembler discussed in *Dr. Dobb's Toolbook of 68000 Programming,* complete with source code and documentation. The Cross-Assembler requires CP/M 2.2 with 64K or MS-DOS with 128K. The disk is available in the following formats: MS-DOS, 8" SS/SD, and Osborne.

Interfacing to S-100/IEEE 696 Microcomputers
Mark Garetz and Sol Libes
Item #85-2 $24.95
This book helps S-100 bus users expand the utility and power of their systems. It describes the S-100 bus with unmatched precision. Various chapters describe its mechanical and functional design, logical and electrical relationships, bus interconnections, and busing techniques.

General Interest

Public-Domain Software: Untapped Resources for the Business User
Rusel DeMaria and George R. Fontaine
Item #39-9 $19.95 (book)
Item #47-X $34.95 (book/disk)
Organized into a comprehensive reference, this book introduces the novice and guides the experienced user to a source of often overlooked software—public domain and Shareware. This book will tell you where it is, how to get it, what to look for, and why it's for you. The sample programs and some of the software reviewed is available on disk in MS-DOS format. Includes $15 worth of free access time on CompuServe!

Dr. Dobb's Journal Bound Volume Series

Each volume in this series contains a full year's worth of useful code and fascinating history from *Dr. Dobb's Journal of Software Tools*. Each volume contains every issue of *DDJ* for a given year, reprinted and combined into one comprehensive reference.

Volume	1: 1976	*Item #13-5*	*$30.75*
Volume	2: 1977	*Item #16-X*	*$30.75*
Volume	3: 1978	*Item #17-8*	*$30.75*
Volume	4: 1979	*Item #14-3*	*$30.75*
Volume	5: 1980	*Item #18-6*	*$30.75*
Volume	6: 1981	*Item #19-4*	*$30.75*
Volume	7: 1982	*Item #20-8*	*$35.75*
Volume	8: 1983	*Item #00-3*	*$35.75*
Volume	9: 1984	*Item #08-9*	*$35.75*
Volume	10: 1985	*Item #21-6*	*$35.75*
Volume	11: 1986	*Item #72-0*	*$35.75*
Volume	12: 1987	*Item #84-4*	*$39.95*

To order any of these products send your payment, along with $2.25 per item for shipping, to M&T Books, 501 Galveston Drive, Redwood City, California 94063. California residents, please include the appropriate sales tax. Or, call toll-free 800-533-4372 (in California 800-356-2002) Monday through Friday between 8 A.M. and 5 P.M. PST. When ordering disks, please indicate format.

THE PROGRAMMER'S Essential OS/2 HANDBOOK

DAVID E. CORTESI

If you write programs for OS/2, you'll never want to let this book get more than an arm's reach away from your keyboard. ***The Programmer's Essential OS/2 Handbook*** succinctly organizes the many features of OS/2 into related topics and illuminates their uses.

The text is woven together by two indexes and a web of cross-references. Wherever a paragraph on one topic depends on another topic, there's a cross-reference.

You'll find:

- An overview of OS/2's architecture and important terms
- How the CPU operates in Protect mode, and how OS/2 uses it
- Exhaustive coverage of program linking and loading
- How programs get information about the system
- How programs handle hardware traps and software signals
- How subordinate programs are loaded and controlled
- A detailed look at how asynchronous tasks coordinate their work
- How to exploit the features of the file system
- How to program the screen, keyboard, mouse, and serial port
- How to build a monitor for the keyboard or printer

More than 200 system functions are laid out alphabetically. The calling sequence for each is shown graphically and in declarations for C and Pascal. The operation of each function is described in terse, unambiguous language, and common error codes are listed.

All sample programs and source code are available on disk.

M&T Publishing, Inc.
501 Galveston Drive, Redwood City, CA 94063

M&T BOOKS

ISBN 0-934375-82-8 >$24.95